Lignes de fracture dans les pavages par dominos.

1 Introduction

Dans ce document, on cherche à montrer que l'ensemble des pavages d'un graphe (défini ci-après) peut être muni d'une structure de treillis qui peut elle-même être factorisée.

La deuxième partie introduit le graphe étudié ainsi que le formalisme utilisé, qui a été choisi pour traiter le cas le plus général possible.

Les deux parties suivantes présentent les outils utilisés et la structure de treillis [1]. Les résultats proviennent majoritairement de la thèse de M. Sébastien DESREUX [3] et ont été adaptés pour être cohérents avec les fonctions d'équilibre qui permettent d'étendre l'étude aux domaines avec trous et qui sont définies dans l'article *Domino tilings and related models : space of configurations of domains with holes* [4].

La cinquième partie aboutit au théorème de fracture. Les définitions et résultats proviennent également de la thèse (hormis la proposition 3) et ont presque tous été réécrits pour conserver la généralité du propos.

L'essentiel de la sixième partie est constituée de conjectures, vraies dans le cas des domaines sans trou.

La dernière partie expose brièvement le programme, écrit en $C++$ [2], qui permet de calculer une fonction d'équilibre, les pavages minimaux et maximaux ainsi que les lignes de fracture d'un domaine dans une grille carrée.

Ce travail a été réalisé seul.

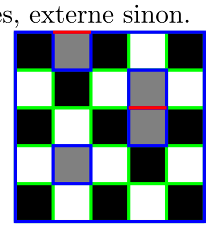
2 Définitions générales

On définit ici le type de graphe étudié et les notations qui seront utilisées dans le reste du document. Le formalisme utilisé sert à traiter le cas le plus général possible. En pratique on étudie la grille carrée ou la grille triangulaire du plan.

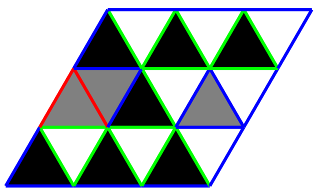
On se place dans le plan $\mathbb{R}^2$.
G = (X, A) désigne un graphe planaire non orienté ($X \subset \mathbb{R}^2$, une arête (v, v') $\in$ A correspond au segment $[v, v']$) tel que tout point de X appartient à un cycle de G et tel que les arêtes ne se croisent pas.

Tous les chemins de G considérés ne passent qu'une et une seule fois par chaque sommet et arête rencontrés.
On appelle cellule de G toute région (fermée) du plan délimitée par un cycle de G dont l'intérieur ne contient aucune arête.
On suppose que le graphe formé à partir des cellules de G est 2-coloriable (et on le considère colorié en noir et blanc) et que toutes les cellules ont le même nombre de côtés n_c.
On peut considérer un ensemble de cellules (éventuellement vide), dites interdites, que l'on colorie en gris.

Une arête de G est alors dite interne si elle délimite deux cellules non interdites de G, interdite si elle délimite deux cellules interdites, externe sinon.



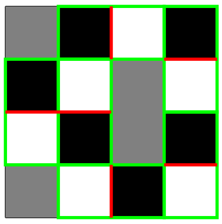
Exemple de domaine dans une grille carrée



Exemple de domaine dans une grille triangulaire

Le domaine R de G est la réunion des cellules de G et ∂R est la frontière de R (qui est la réunion des arêtes externes). On peut supposer R connexe par arcs quitte à travailler sur chaque composante connexe.

Un pavage P de G est un sous-ensemble de A tel que pour toute cellule non interdite c de G, toutes les arêtes qui délimitent c sont dans P sauf exactement une, on appelle tuile de P toute cellule de (X, P).



Exemple de pavage dans un domaine d'une grille carrée

Définition (cellule enclose par un cycle)**.** *Une cellule de G est enclose dans un cycle de G si et seulement si tout chemin liant un sommet de la cellule à la bordure de G contient un sommet du cycle.*

Définition (cycle horaire)**.** *Soit C un cycle de G. C est horaire si en parcourant C, les cellules de droite sont encloses dans C. Sinon C est anti-horaire.*

Définition (déséquilibre d'un cycle)**.** *Soit C un cycle de G. Si N et B désignent le nombre de cellules noires et blanches encloses par C, alors le déséquilibre de C est :* $Des(C) = N - B$ *si C est horaire* $Des(C) = B - N$ *sinon*

3 Fonctions de hauteur

On introduit ici les fonctions de hauteur, qui sont l'outil permettant de travailler sur les pavages. On pourra en première lecture considérer le cas d'un domaine sans trou et ignorer les fonctions d'équilibre (eq = 0).

On suppose G pavable et on fixe un point $v_0 \in X$ jusqu'à la section 5.

Définition (spin)**.** *Soit la fonction* $sp : \begin{array}{ccc} A & \longrightarrow & \{-1,1\} \\ a & \longmapsto & sp(a) \end{array}$ *définie par : Pour a = (v, v') ∈ A, si en suivant l'arête a de v à v', la cellule de gauche est noire, sp(a) = 1, sinon sp(a) = -1. On dit que a est positive si sp(a) = 1, négative sinon.*

Remarque. $\forall (v, v') \in A, sp((v, v')) = -sp((v', v))$

Remarque. *Soient* $f : A \longrightarrow \mathbb{Z}$ *et* $C = (v_0, ..., v_n)$ *un chemin de G. On pose :* $f(C) = \sum_{i=0}^{n-1} f(v_i, v_{i+1})$

Remarque. *Dans un domaine sans trou, on a pour tout cycle C :* $n_c Des(C) = sp(C)$*, ce qui motive la définition suivante.*

Définition (fonction d'équilibre)**.** $eq : A \longrightarrow \mathbb{Z}$ *est une fonction d'équilibre si et seulement si :* $\forall (v, v') \in A, eq(v, v') = -eq(v', v)$ *Pour tout cycle* $C, n_c \times Des(C) = eq(C) + sp(C)$

Remarque. *On dispose d'un algorithme pour calculer une fonction d'équilibre et on en considère une qu'on notera dans toute la suite eq.*

Définition (différence de hauteur)**.** *Soit* $C = (v_1, v_2, ..., v_n)$ *un chemin de G. La différence de hauteur de C est :* $\Delta h(C) = eq(C) + sp(C)$

Remarque. *Si C est un cycle, on a* $\Delta h(C) = Des(C)$*.*

Définition (chemin valide dans P)**.** *Soit P un pavage de G. Un chemin* $C = (v_1, v_2, ..., v_n)$ *est dit valide dans P si et seulement si :* $\forall 1 \leq k \leq n-1, (v_k, v_{k+1}) \in P$ *Autrement dit, un chemin valide dans P est un chemin qui ne coupe pas de tuile de P.*

Lemme 1. *Soit P un pavage de G. La différence de hauteur d'un cycle valide dans P est nulle*

Définition (fonction de hauteur)**.** *Soit P un pavage de G. Pour tout sommet v de G, soit C(v) un chemin valide dans P de* v_0 *à v. La fonction de hauteur introduite par P est* $h_P : \begin{array}{ccc} X & \longrightarrow & \mathbb{Z} \\ v & \longmapsto & \Delta h(C(v)) \end{array}$.

Remarque. *Les fonctions de hauteur introduites par Thurston sont un élément essentiel de l'étude des pavages. Le lemme précédent permet d'assurer que cette définition suivante est correcte, puisque la différence de hauteur entre deux nœuds ne dépend pas du chemin suivi.*

Proposition 1. *Soit P un pavage de G, soit* $(v, v') \in A$ *positive. Alors :* $h_P(v') - h_P(v) - eq(v, v') \in \{-n_c + 1, 1\}$

Théorème 1. *Les fonctions de hauteur sont exactement les fonctions vérifiant le proposition 1 et prenant la valeur 0 en* v_0*.*

Les pavages sont en bijection avec les fonctions de hauteur.

Remarque. *Ce théorème est fondamental car il ramène l'étude des pavages à celle, plus simple, des fonctions de hauteur et caractérise entièrement ces dernières.*

4 Le treillis des pavages

On voit ici comment les fonctions de hauteur permettent de munir les pavages d'une structure de treillis.

4.1 Les treillis

On considère (E, $\preceq$) un ensemble non vide (partiellement) ordonné.

Définition (infimum, supremum)**.** *Soit F une partie de E non vide. L'infimum et le supremum de F sont (s'ils existent) : $max(\{y \in E, \forall x \in F, y \preceq x\})$ et $min(\{y \in E, \forall x \in F, x \preceq y\})$*

Définition (treillis)**.** *E est un treillis si toute partie non vide de E admet un infimum et un supremum.*

On suppose maintenant que E est un treillis.

Remarque. *Pour tous $x, y \in E$, on note $x \wedge y$ l'infimum de $\{x, y\}$ et $x \vee y$ son supremum.*

Définition (treillis distributif)**.** *E est distributif si et seulement si pour tous $x, y, z \in E$: $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$*

Proposition 2 (treillis fini)**.** *Supposons E fini. Alors E est un treillis si et seulement si pour tous $x, y \in E$, $x \wedge y$ et $x \vee y$ existent.*

4.2 Le cas des pavages

Définition (ordre sur l'ensemble des pavages)**.** *Soient P_1, P_2 deux pavages de G. On dit que P_1 est inférieur à P_2 et on note $P_1 \preceq P_2$ si et seulement si $h_{P_1} \leq h_{P_2}$.*

Lemme 2. *Soient P_1 et P_2 deux pavages de G. On a : $\forall v \in X, n_c \mid h_{P_1}(v) - h_{P_2}(v)$*

Théorème 2 (min et max de fonctions de hauteur)**.** *Soient P_1 et P_2 deux pavages de G. Alors $h_{min} = min(\{h_{P_1}, h_{P_2}\})$ et $h_{max} = max(\{h_{P_1}, h_{P_2}\})$ sont des fonctions de hauteur.*

Corollaire. *L'ensemble des pavages muni de la relation d'ordre $\preceq$ est un treillis distributif.*

Définition. *Il existe un pavage minimal et un pavage maximal, dont on note les fonctions de hauteur h_{min} et h_{max}.*

5 Théorème de fracture

Cette structure de treillis peut être factorisée en treillis produit grâce au théorème de fracture, qui est l'objectif de cette section.

On peut reprendre l'étude précédente pour n'importe quel sommet de G pour obtenir des fonctions de hauteur relatives à un sommet s, qu'on notera avec un indice s.

Définition (sous-domaine)**.** *Un sous-domaine D de R est une réunion de cellules de G pavable.*

Définition (ligne de fracture)**.** *Une ligne de fracture de G est un chemin de G valide dans tout pavage de G.*

Définition (point solide relativement à un sommet s)**.** *Soient $v, s \in X$. v est dit solide relativement à s si et seulement si $h_{s,max}(v) = h_{s,min}(v)$. v a alors la même hauteur dans tout pavage de G, notée $h_s(v)$.*
v est dit solide s'il est solide relativement à un sommet quelconque de G (sauf v)

Remarque. *Quant on introduit un point v solide, on peut introduire v_0 tel que v est solide relativement à v_0 et reprendre les notations des parties précédentes.*

Le lien entre les lignes de fracture et les points solides est exprimé par la propriété suivante :

Proposition 3. *Les relations :*
— $\forall x, y \in X, x\mathcal{R}y \Leftrightarrow$ *il existe une ligne de fracture passant par x et y*
— $\forall x, y \in X, x\mathcal{R}'y \Leftrightarrow$ *y est solide relativement à x*
sont des relations d'équivalence identiques.

Définition (sous-domaine prometteur)**.** *Un sous-domaine D de G est dit prometteur si et seulement si aucun sommet à l'intérieur de D n'appartient à une ligne de fracture cyclique.*

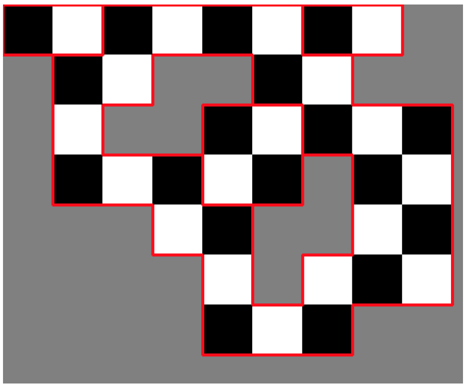
Définition (union fertile)**.** *soient D_1 et D_2 deux sous-domaines prometteurs de G. On dit que $D_1 \cup D_2$ est fertile si et seulement si $D_1 \cup D_2$ est un sous-domaine prometteur de G.*

Définition (zone fertile)**.** *Une zone fertile de G est un sous-domaine prometteur maximal pour l'inclusion fertile.*

Proposition 4. *Un sous-domaine prometteur de G est fertile si et seulement si sa frontière est formée de lignes de fracture.*

Définition (zone de fracture)**.** *Soit D un sous-domaine de G. D est une zone de fracture si et seulement si D a un point dans son intérieur, ∂D est une ligne de fracture et D ne contient aucune autre ligne de fracture cyclique que sa frontière.*

Théorème 3 (de fracture)**.** *Les zones de fracture de G sont d'intérieurs disjoints.*
Le treillis des pavages de G est le produit des treillis des zones de fracture.



Lignes de fracture et zones de fracture dans un domaine à trous.

6 Résultats complémentaires

Proposition 5. *Soit $v \in X - \partial R$. Si v est solide, alors il existe au moins une arête positive et une arête négative partant de v valides dans tout pavage de G.*

Le résultat suivant permet d'accélérer le calcul des lignes de fracture puisqu'il suffit de calculer les pavages minimal et maximal relativement à un sommet de chaque amas connexe de trous et non plus de chaque sommet.

Conjecture 1. *Pour $x \in X$, il existe $y \in X \cap \partial R$ solide relativement à x.*

Le résultat suivant permet d'alléger significativement la fin de la quatrième partie.

Conjecture 2. *soient $x \in X$ et $y \in X - \{x\}$ solide relativement à x. Alors y appartient à une ligne de fracture cyclique.*

Conjecture 3. *La factorisation du treillis des pavages de G par les lignes de fracture est optimal, c'est-à-dire que le treillis des pavages d'une zone de fracture n'est pas décomposable en produit de treillis non triviaux.*

Remarque. *Cette conjecture est vraie dans le cas où aucune cellule n'est interdite.*

7 Mise en œuvre

Le code source, sur lequel s'appuie cette section, est donné en annexe.

L'objectif de cette section est de réaliser une analyse de la complexité du calcul de la fonction d'équilibre, des pavages minimaux et maximaux et des lignes de fracture.

On notera $|E|$ le cardinal d'un ensemble E ou d'un conteneur quelconque ou le nombre de sommets de degré non nul si E est un graphe. On peut d'ores et déjà remarquer que : $|A| = \bigcirc(|X|)$ (car $|A| \leq 4|X|$).

7.1 Commentaires sur le code source

L'interface graphique a été réalisée à l'aide de la bibliothèque SDL2. Les fichiers Application.h, Window.h et Screenshot.h sont destinés à simplifier l'utilisation de cette bibliothèque et ne concernent donc pas directement le sujet étudié.

De même les fichiers Graphe.h, ArbreAVL.h, Chaine.h et UnionFind.h ne sont pas indispensables à la compréhension du programme. Le code de ces quatre fichiers a été écrit de manière générique, c'est-à-dire pour s'appliquer au plus de cas possibles. Par exemple, le coloriage des sommets d'un graphe n'est pas utilisé dans ce programme. L'utilisation des arbres AVL n'est pas indispensable pour obtenir une bonne complexité mais sert pour les graphes denses (et permet d'obtenir un tri rapide et facile à écrire).

7.2 Complexité

Si f est une fonction, on note $c(f)$ sa complexité.

On s'intéresse à $c(equilibrage)$:
Le calcul des amas de trous se fait en $\bigcirc(n)$
Puis le tracé des lignes verticales reliant les amas se fait en $\bigcirc(n\sqrt{|trous|})$.
Le calcul des valuations s'exécute en $\bigcirc(|X|)$.
Donc : $c(equilibrage) = \bigcirc(|X| + n\sqrt{|X|}) + c(equilibreTrou)$
La fonction récursive equilibreTrou est appelée exactement $|tetesTrous|$ fois puisque par construction, tetesTrous est un arbre connexe. Les exécutions successives de "tetesTrous.suivants(trou)" renvoient au total deux fois chaque sommet de degré non nul de tetesTrous pour un coût global en $\bigcirc(|tetesTrous|)$.
Le calcul effectif de la fonction d'équilibre s'effectue à chaque appel en $\bigcirc(\sqrt{|X|})$ pour un coût total en $\bigcirc(|tetesTrous|\sqrt{|X|})$.
Or (de retour dans equilibrage) $|tetesTrous| \leq |trous| + 1$, donc on obtient finalement :
$c(equilibrage) = \bigcirc(|X| + |trous|\sqrt{|X|})$.

Les calculs étant les mêmes (aux rôles des tableaux min et max près), on a
$c(pavageCarreMin) = c(pavageCarreMax)$. On peut alors se contenter d'étudier pavageCarreMin.
L'initialisation de min, max et hauteur se fait en $\bigcirc(|X|)$.
Puis le parcours en largeur d'arbre s'exécute en $\bigcirc(|X|)$
La récupération des sommets dont il faut augmenter la hauteur a encore une complexité en $\bigcirc(|X|)$
Chaque passage dans la dernière boucle while s'effectue en $\bigcirc(1)$ puisque chaque sommet est de degré au plus 4. De plus, la quantité $\sum_{i=0}^{|X|-1} hauteur[i]$ augmente de 4 en 4 de $\sum_{i=0}^{|X|-1} min[i]$ jusqu'à au plus $\sum_{i=0}^{|X|-1} max[i]$. Donc il y a au plus $\frac{1}{4}\sum_{i=0}^{|X|-1} max[i] - min[i]$ passages dans cette boucle while. Soient $v \in X$ et C un chemin de v_0 à v dans arbre. Alors $max(v) - min(v) = 4|C|$ (par récurrence sur $|C|$ avec le calcul effectué dans la première boucle while). Donc $max(v) - min(v) \leq 4|A| \leq 16|X|$. Par conséquent, on passe au plus $4|X|^2$ fois dans la boucle while, ce qui donne une complexité en $\bigcirc(|X|^2)$.
Donc $c(pavageCarreMin) = \bigcirc(|X|^2)$.

Pour l'affichage des lignes de fracture, on a $c(dispFractures) = \bigcirc(|X|)$. Donc en admettant la conjecture 1, on obtient un coût total en $\bigcirc(|amasTrous||X|^2)$ (à cause des pavages), sinon en $\bigcirc(|X|^3)$.

Références

[1] Pierre Boutillier. Une classe d'ordres partiels : les treillis, 2009. http ://perso.ens-lyon.fr/eric.thierry/Graphes2009/pierre-boutillier.pdf.

[2] Claude Delannoy. *Programmer en language C++ (8e édition).* EYROLLES, 2014.

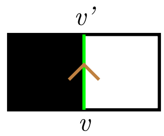
[3] Sébastien Desreux. *Aspects algorithmiques de la génération de pavages.* PhD thesis, Université Paris 7, 2003.

[4] Sébastien Desreux, Martin Matamala, Ivan Rapaport, and Eric Rémila. Domino tilings and related models : space of configurations of domains with holes. *Elsevier Science*, pages 3–18, 2003.

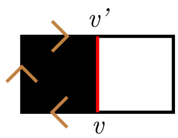
A Démonstrations

Démonstration (Lemme 1)**.** *Si C est un cycle valide dans P, alors il enclôt un nombre entier de tuiles qui contiennent toutes une cellule noire et une cellule blanche. Donc C enclôt autant de cellules noires que de cellules blanches, d'où :* $\Delta h(C) = eq(C) + sp(C) = n_c \times Des(C) = 0$

Démonstration (Proposition 1)**.** *Si $(v,\ v')$ est valide dans P,* $h_P(v') - h_P(v) = eq(v, v') + sp(v, v') = 1 + eq(v, v')$.



Sinon, soit $C = (v, v_3, ..., v_{n_c}, v', v)$ un cycle horaire qui n'enclôt que la cellule à gauche de $(v,\ v')$. C est valide dans P et composé de n_c arêtes négatives. Alors : $h_P(v') - h_P(v) = eq(v, v_3, ..., v_{n_c}, v') + sp(v, v_3, ..., v_{n_c}, v') = eq(C) + sp(C) + sp(v, v') + eq(v, v') = -n_c + 1 + eq(v, v')$.



Démonstration (Théorème 1)**.** *Soit $h : X \longrightarrow \mathbb{Z}$ vérifiant le proposition 1 et telle que $h(v_0) = 0$. Soit* $G: \begin{array}{ccc} A & \longrightarrow & \mathbb{Z} \\ (v, v') & \longmapsto & h(v') - h(v) - eq(v, v') \end{array}$. *Considérons une cellule de G et C un cycle, horaire si la cellule est noire, anti-horaire sinon, qui n'enclôt que cette cellule. On a $eq(C) = 0$, donc $G(C) = 0$, et alors il y a une unique arête a sur le bord de la cellule vérifiant $G(a) = -n_c + 1$. En retirant uniquement les arêtes de G vérifiant cette condition, on obtient un pavage T de G.*
Montrons que $h = h_T$. Soit $v_1 \in X$ et soit C un chemin de v_0 à v_1 valide dans T. Alors par construction pour toute arête $(v,\ v')$ de C, $h(v') - h(v) = h_T(v') - h_T(v) = sp(v, v') - eq(v, v')$, donc $h(v_1) = h_T(v_1)$.

Démonstration (Lemme 2)**.** *Par récurrence sur les nœuds en partant de v_0 avec le proposition 1.*

Démonstration (Théorème 2)**.** *Le cas de h_{max} est analogue à celui de h_{min}.*
Montrons que pour toute arête $(v,\ v')$ positive, $h_{min}(v') - h_{min}(v) - eq(v, v') \in \{-n_c + 1, 1\}$.
Si $h_{P_1}(v) = h_{P_2}(v)$, alors par $h_{min}(v') = h_{P_1}(v')$ ou $h_{min}(v') = h_{P_2}(v')$ et le proposition 1 donne le résultat.
Sinon, on peut supposer $h_{P_1}(v) < h_{P_2}(v)$. Alors d'après le lemme 2, $h_{P_1}(v) \leq h_{P_2}(v) - n_c$. Donc d'après le proposition 1, $h_{P_1}(v') \leq h_{P_1}(v) + eq(v, v') + 1 \leq h_{P_2}(v) + eq(v, v') + 1 - n_c \leq h_{P_2}(v')$. D'où le résultat encore avec le proposition 1.

Démonstration (Corollaire)**.** *La structure de treillis découle du théorème précédent. La distributivité se réduit à celle des valeurs prises par les fonctions de hauteur, c'est-à-dire à la distributivité sur les entiers.*

Démonstration (Proposition 3)**.**

Reflexivité *Soit $x \in X$. Alors (x) est une ligne de fracture de G et x est solide relativement à x*

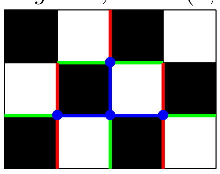
Symétrie *Soient $x, y \in X$. Supposons $x\mathcal{R}y$. On dispose d'une ligne de fracture C passant par x et y. Alors C passe par y et x donc $y\mathcal{R}x$.*
Supposons que y est solide relativement à x. Soit h la hauteur de y dans tout pavage de G lorsque x a 0 pour hauteur. Alors, dans tout pavage de G où y a 0 pour hauteur, x a la hauteur $-h$ donc x est solide relativement à y

Transitivité *Soient $x, y, z \in X$. Supposons qu'on ait deux lignes de fracture passant respectivement par x et y et par y et z. On peut ne considérer que des restrictions de ces chemins commençant respectivement par x et y et se terminant respectivement par y et z. Alors la concaténation des deux chemins obtenus est une ligne de fracture allant de x à z.*
Supposons que y est solide relativement à x et que z est solide relativement à y. Soit h (resp. g) la hauteur de y (resp. z) lorsque la hauteur de x (resp. y) est 0. Alors dans tout pavage de G, lorsque x est à la hauteur 0, z est à la hauteur $g - h$. Donc z est solide relativement à x.

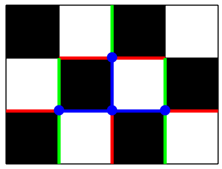
Donc $\mathcal{R}$ et $\mathcal{R}'$ sont des relations d'équivalence.

Montrons maintenant que ces relations sont identiques : Soient $x, y \in X$, supposons qu'on dispose d'une ligne de fracture passant par x et y. On peut considérer $C = (x, ..., y)$ une restriction de ce chemin. Alors dans tout pavage de G, lorsque x est à la hauteur 0, y est à la hauteur $sp(C) + eq(C)$. Donc y est solide relativement à x.
Soit $x \in X$. On va montrer que la classe d'équivalence de x pour la relation $\mathcal{R}'$ est incluse dans la classe d'équivalence de x pour la relation $\mathcal{R}$, qu'on notera $\overline{x}$.
On prend $v_0 = x$ et on revient aux notations des parties précédentes.
Soit $y \in X - \overline{x}$ ayant un voisin $z \in \overline{x}$. La hauteur de z est la même dans tout pavage de G donc la hauteur de y ne peut prendre que deux valeurs selon que (z, y) est valide ou non.

- *si (z, y) est positive, alors (z, y) est invalide dans P_{min} et valide dans P_{max}*
- *si (z, y) est négative, alors (z, y) est valide dans P_{min} et invalide dans P_{max}*



Dans le pavage minimal



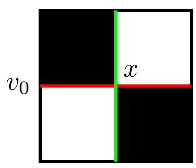
Dans le pavage maximal

Alors en partant de P_{min} et en rendant valides les arêtes invalides, et invalides les arêtes valides du type (z, y) où $y \in X - \overline{x}$ et $z \in \overline{x}$, on obtient un pavage P tel que $\forall y \in X - \overline{x}, h_P(y) = h_{min}(y) + n_c$.
Donc aucun sommet de $X - \overline{x}$ n'est solide relativement à x.
Par contraposition, tout sommet solide relativement à x est dans $\overline{x}$, ce qui est bien ce qu'on voulait montrer.
Donc les relations $\mathcal{R}$ et $\mathcal{R}'$ sont identiques.

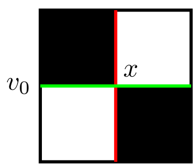
Démonstration (Proposition 4)**.** *Notons D ce sous-domaine prometteur. Supposons D fertile. S'il y a un sommet v de la frontière qui n'est pas solide, alors $v \notin \partial R$. Soit P un pavage de G, considérons H la réunion des tuiles qui entourent v. Alors H est prometteur, or D est fertile, donc $D \cup H$ a un sommet intérieur solide qui participe à une ligne de fracture cyclique, qui ne peut être que v. Donc v est solide*
Supposons que ∂D est formée de lignes de fracture. Soit H un sous-domaine prometteur de G non inclus dans H et tel que $D \cup H$ soit un sous-domaine de G. Alors l'un des sommets de $D \cup H$ est un sommet de ∂D, donc $D \cup H$ n'est pas prometteur (car ∂D est formée de cycles).
Donc D est fertile.

Démonstration (Théorème 3)**.** *Si deux zones de fracture étaient d'intérieurs non disjoints, alors leur union serait un domaine prometteur, ce qui est absurde puisque les zones de fracture sont des zones fertiles.*
G est donc l'union disjointe de ses zones de fracture. Tout pavage de G pave les zones de fracture de G et la donnée pour chaque zone de fracture de G, d'un pavage de cette zone de fracture donne un pavage de G. Donc tout pavage de G est la donnée des pavages des zones de fracture.

Proposition 6 (flip)**.** *Soit $x \in X - \partial R$, soit $v_0 \in X - \{x\}$.*
Soit P un pavage de G tel que toutes les arêtes positives partant de x soient invalides dans P. Alors le flip ascendant en x est l'opération qui rend valides toutes les arêtes positives et invalides les arêtes négatives. Le flip ascendant en x construit un nouveau pavage P' tel que $\forall y \in X - \{x\}, h_{P'}(y) = h_P(y)$ et $h_{P'}(x) = h_P(x) + n_c$.
Soit P un pavage de G tel que toutes les arêtes négatives partant de x soient invalides dans P. Alors le flip descendant en x est l'opération qui rend valides toutes les arêtes négative et invalides les arêtes positives. Le flip descendant en x construit un nouveau pavage P' tel que $\forall y \in X - \{x\}, h_{P'}(y) = h_P(y)$ et $h_{P'}(x) = h_P(x) - n_c$.



Après un flip descendant



Après un flip ascendant

Remarque. *Le passage d'un maximum à un minimum local (ou inversement) s'appelle un flip.*

Démonstration (Proposition 5)**.** *Soit $v_0 = v$.*
Supposons qu'aucune arête positive partant de x n'est valide dans tout pavage de G.
Soit $(x, v) \in A$ une arête positive et soit P un pavage dans lequel (x, v) est invalide. On a d'après la proposition 1 : $h_P(v) \in \{eq(x, v) - n_c + 1, eq(x, v) + 1\}$ (car $h_P(x) = 0$). Or $h_P(v) = eq(x, v) - n_c + 1$ donc dans le pavage

minimal de G : $h_{min}(v) = eq(x,v) - n_c + 1$, c'est-à-dire l'arête (x,v) est invalide dans P_{min}.
Donc toute arête positive partant de x est invalide dans P_{min}.
Soit maintenant $v_0 \in X - \{x\}$.
Alors en reprenant le pavage minimal précédent, on peut réaliser un flip ascendant sur x.
Donc x n'est pas solide.
Par contraposition, si x est solide, alors on dispose d'une arête positive partant de x qui est valide dans tout pavage de G.
En raisonnant de même avec le pavage maximal on obtient une arête négative valable dans tout pavage de G.

Démonstration (Conjecture 1, partielle). *Construisons une suite $(x_n)_{n\in\mathbb{N}} \in X^{\mathbb{N}}$ récursivement :*

— $x_0 = x$

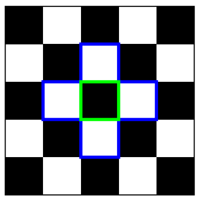
— *supposons x_n construit, $n \in \mathbb{N}$. Si $x_n \in \partial R$, on arrête la construction, sinon on prend x_{n+1} solide relativement à x_n et tel que (x_n, x_{n+1}) est positive (ce qui est possible d'après la proposition précédente)*

Si la construction précédente s'achève à l'indice $n \in \mathbb{N}$, alors en prenant $y = x_n$, on a $y \in X \cap \partial R$ et par transitivité de la relation "être solide relativement à", y est solide relativement à x.

Sinon, soient $n = inf(\{n \in \mathbb{N}^*, x_n \in \{x_0, ..., x_{n-1}\}\})$ (l'ensemble précédent est non vide d'après le principe des tiroirs) et $k \in [\![0, n-1]\!]$ tel que $x_k = x_n$. Alors le domaine délimité par $C = (x_k, ..., x_n)$ est pavable. Or un tel domaine n'est pas pavable, ce qui constitue une contradiction, donc la construction précédente est finie.

Il reste à montrer que le domaine délimité par C n'est pas pavable.

Pour ce faire, une idée peut être de considérer l'ensemble A des cellules encloses par C qui ont une arête de leur bordure dans C et l'ensemble B des cellules encloses dans C qui partagent une arête de leur bordure avec une cellule de A et qui ne sont pas dans A. On remarque alors que si les cellules de A sont noires, alors celles de B sont blanches et inversement. Dans un éventuel pavage du domaine délimité par C, comme toutes les arêtes de C sont valides, chaque cellule de A partage une arête invalide avec une cellule de B. Il suffit alors de montrer que : $Card(B) < Card(A)$.



En bleu le cycle C, en vert le cycle séparant les cellules de A de celles de B

En remarquant que les arêtes séparant les cellules de A et celles de B forment un cycle, on peut alors chercher à montrer le résultat suivant :
soit C un cycle, soit B, (resp. A) l'ensemble des cellules encloses (resp. non encloses) par C et ayant une arête de leur bordure dans C. Alors $Card(B) < Card(A)$ (on semble même avoir $Card(B) \leq Card(A) + n_c - 1$).

B Code source

B.1 main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "Application.h"
#include "Window.h"
#include "Screenshot.h"
#include "Graphe.h"
#include "ArbreAVL.h"
#include "Chaine.h"
#include "UnionFind.h"
// la conjecture 1 permet de ne calculer les lignes de fracture en ne faisant les calculs que pour un sommet par amas de trous
#define CONJ1
//ces variables sont declarees globales car elles seront utilisees tout au long du programme
const int T_FEN = 803;
const int T_CASE = 50;
const int NB_NOEUDS_COTE = ((T_FEN-3)/T_CASE)+1;
const int NB_NOEUDS = (NB_NOEUDS_COTE*NB_NOEUDS_COTE);
const int NB_CASES = ((NB_NOEUDS_COTE-1)*(NB_NOEUDS_COTE-1));
```

```
const double PROPORTION_TROUS = 1/5.0;
const bool imposeTrous = true;
Chaine<unsigned int> trousImposes({48, 32, 17, 35, 257, 241, 5, 23, 11, 27, 102, 86, 262, 244, 168, 184});
//Chaine<unsigned int> trousImposes({8, 24, 25, 43, 60, 77, 94, 110, 109, 125, 124, 123, 105, 88, 70, 69, 51, 34, 17, 90, 73, 74, 57, 21, 20, 37, 36});
//Chaine<unsigned int> trousImposes({97, 79, 126, 144, 160, 73, 106, 90, 55, 71});
//Chaine<unsigned int> trousImposes({3, 170, 171, 172, 173, 174, 175, 176, 178, 179, 180, 181, 182, 183, 184, 185/*, 161, 142, 143*/});
//Chaine<unsigned int> trousImposes({6, 23, 40, 57, 74, 91, 108, 125, 124, 123, 122, 121, 120, 119, 18, 20, 21, 35, 37, 38, 52, 55, 69, 72, 86, 87, 88, 89});
//Chaine<unsigned int> trousImposes({6, 23, 40, 57, 56, 55, 72, 71, 70, 69, 68, 51, 34,17, 18, 20, 21});
//Chaine<unsigned int> trousImposes({36, 70, 105, 139, 8, 25, 42, 59, 76, 93, 110, 127, 144, 161, 160, 159, 158, 157, 156, 155, 154, 153});
// les algorithmes de calcul de l'equilibre et des pavages minimaux et maximaux sont ceux presentes dans l'article Domino tilings and related models: space of configurations of domains with holes (a quelques revisions pres)
/* struct CA : structure des couleurs des aretes des noeuds des graphes etudies */
typedef struct CA {
    int spin;
    int equilibre;
    char bordure;
    CA(int s = 0, int e = 0, bool bo = 0) : spin(s), equilibre(e), bordure(bo) {}
} CA;
/* void treeSort(Chaine<T> &c) : trie la chaine c par ordre croissant. Il faut que l'objet T dispose de l'operateur < */
template<typename T>
void treeSort(Chaine<T> &c) {
    ArbreAVL<T> a;
    typename Chaine<T>::Iterator it(c, 0);

    while(!it.end()) {
        a.addElem(*it);
        ++it; }
    c.vider();
    typename ArbreAVL<T>::Iterator itA(a);
    while(!itA.end()) {
        c.addElem(-1, *itA);
        ++itA; }
    c.reverse();
}
/* void drawGrilleCarree(Window &fen) : dessine une grille carree de cote (NB_NOEUDS_COTE-1)*T_CASE sur la FenGrapheetre FenGraphe */
void drawGrilleCarree(Window &fen) {
    int i, j;

    fen.setColor(255, 255, 255);
    fen.drawRect(0, 0, fen.w(), fen.h());
    fen.setColor(0, 0, 0);
    //au moins 1 pixel de marge en haut et a gauche, 2 en bas et a droite
    for(i = 0; i < NB_NOEUDS_COTE-1; ++i)
        for(j = i%2; j < NB_NOEUDS_COTE-1; j += 2)
            fen.drawRect(i*T_CASE+1, j*T_CASE+1, T_CASE, T_CASE);
    i = 1+(NB_NOEUDS_COTE-1)*T_CASE;
    fen.drawLine(1, 1, i, 1); //dessin du cadre
    fen.drawLine(1, 1, 1, i);
    fen.drawLine(i, 1, i, i);
    fen.drawLine(1, i, i, i);
}
/* void relierCaseCarree(Window &fen, int lig1, int col1, int lig2, int col2) : relie par un trait vert les cases (lig1, col1) et (lig2, col2) sur la FenGrapheetre FenGraphe */
void relierCaseCarree(Window &fen, int lig1, int col1, int lig2, int col2) {
    fen.setColor(0, 0, 255);
    fen.drawLine(1+T_CASE*(col1+0.5), 1+T_CASE*(lig1+0.5), 1+T_CASE*(col2+0.5), 1+T_CASE*(lig2+0.5)); }
/* void makeGrapheCarre(Graphe<char, CA> &g)
 * g est un graphe sans arete, les noeuds sont numerotes de 0 a NB_NOEUDS-1
 * cree les aretes qui font que g devient un graphe carre : tout noeud i est relie exactement aux noeuds qui existent parmi i-1, i+1, i-NB_NOEUDS_COTE, i+NB_NOEUDS_COTE
 * les arcs sont colores par leur spin et leur appartenance a la bordure de g (ceux qui sont sur la bordure de g sont ceux qui relient deux noeuds ayant strictement moins de 4 noeuds adjacents) */
```

```
void makeGrapheCarre(Graphe<char, CA> &g) {
    CA positif(1), negatif(-1);
    int i;

    //noir à gauche <=> positif
    for(i = 0; i < NB_NOEUDS; ++i) {
        if(i%NB_NOEUDS_COTE != 0) { //a gauche
            g.addArc(i, i-1);
            if(((i/NB_NOEUDS_COTE)+(i%NB_NOEUDS_COTE))%2 == 0) g.coloreArc(i, i-1, negatif);
            else g.coloreArc(i, i-1, positif); }
        if(i%NB_NOEUDS_COTE != NB_NOEUDS_COTE-1) { //a droite
            g.addArc(i, i+1);
            if(((i/NB_NOEUDS_COTE)+(i%NB_NOEUDS_COTE))%2 == 0) g.coloreArc(i, i+1, negatif);
            else g.coloreArc(i, i+1, positif); }
        if(i/NB_NOEUDS_COTE != 0) { //en haut
            g.addArc(i, i-NB_NOEUDS_COTE);
            if(((i/NB_NOEUDS_COTE)+(i%NB_NOEUDS_COTE))%2 == 0) g.coloreArc(i, i-NB_NOEUDS_COTE, positif);
            else g.coloreArc(i, i-NB_NOEUDS_COTE, negatif);
        }
        if(i/NB_NOEUDS_COTE != NB_NOEUDS_COTE-1) { // en bas
            g.addArc(i, i+NB_NOEUDS_COTE);
            if(((i/NB_NOEUDS_COTE)+(i%NB_NOEUDS_COTE))%2 == 0) g.coloreArc(i, i+NB_NOEUDS_COTE, positif);
            else g.coloreArc(i, i+NB_NOEUDS_COTE, negatif); } }
    for(i = 0; i < NB_NOEUDS_COTE-1; ++i) {
        positif = g.couleurArc(i, i+1);
        positif.bordure = 1;
        g.coloreArc(i, i+1, positif);

        positif = g.couleurArc(i*NB_NOEUDS_COTE, (i+1)*NB_NOEUDS_COTE);
        positif.bordure = 1;
        g.coloreArc(i*NB_NOEUDS_COTE, (i+1)*NB_NOEUDS_COTE, positif);

        positif = g.couleurArc(NB_NOEUDS_COTE*(NB_NOEUDS_COTE-1) + i, NB_NOEUDS_COTE*(NB_NOEUDS_COTE-1) + i+1);
        positif.bordure = 1;
        g.coloreArc(NB_NOEUDS_COTE*(NB_NOEUDS_COTE-1) + i, NB_NOEUDS_COTE*(NB_NOEUDS_COTE-1) + i+1, positif);

        positif = g.couleurArc((i+1)*NB_NOEUDS_COTE - 1, (i+2)*NB_NOEUDS_COTE - 1);
        positif.bordure = 1;
        g.coloreArc((i+1)*NB_NOEUDS_COTE - 1, (i+2)*NB_NOEUDS_COTE - 1, positif);

        positif = g.couleurArc(i+1, i);
        positif.bordure = 1;
        g.coloreArc(i+1, i, positif);

        positif = g.couleurArc((i+1)*NB_NOEUDS_COTE, i*NB_NOEUDS_COTE);
        positif.bordure = 1;
        g.coloreArc((i+1)*NB_NOEUDS_COTE, i*NB_NOEUDS_COTE, positif);

        positif = g.couleurArc(NB_NOEUDS_COTE*(NB_NOEUDS_COTE-1) + i+1, NB_NOEUDS_COTE*(NB_NOEUDS_COTE-1) + i);
        positif.bordure = 1;
        g.coloreArc(NB_NOEUDS_COTE*(NB_NOEUDS_COTE-1) + i+1, NB_NOEUDS_COTE*(NB_NOEUDS_COTE-1) + i, positif);

        positif = g.couleurArc((i+2)*NB_NOEUDS_COTE - 1, (i+1)*NB_NOEUDS_COTE - 1);
        positif.bordure = 1;
        g.coloreArc((i+2)*NB_NOEUDS_COTE - 1, (i+1)*NB_NOEUDS_COTE - 1, positif); }
}
/* void ajouteTrou(Graphe<char, CA> &g, int n)
 * n identifie le noeud de g qui est au sommet nord-ouest de la case consideree comm un trou
 * etiquette comme faisant partie de la bordure de g toutes les aretes autour de la case
   consideree et retire celles qui etaient deja dans au bord de g (car elles separent deux trous et sont donc invalides */
void ajouteTrou(Graphe<char, CA> &g, int n) {
    CA couleur;

    if(g.existeArc(n, n+1)) {
        couleur = g.couleurArc(n, n+1);
        ++couleur.bordure;
```

```
        if(couleur.bordure == 2) g.removeArc(n, n+1);
        else g.coloreArc(n, n+1, couleur); }
    if(g.existeArc(n, n+NB_NOEUDS_COTE)) {
        couleur = g.couleurArc(n, n+NB_NOEUDS_COTE);
        ++couleur.bordure;
        if(couleur.bordure == 2) g.removeArc(n, n+NB_NOEUDS_COTE);
        else g.coloreArc(n, n+NB_NOEUDS_COTE, couleur); }
    if(g.existeArc(n+NB_NOEUDS_COTE, n+NB_NOEUDS_COTE+1)) {
        couleur = g.couleurArc(n+NB_NOEUDS_COTE, n+NB_NOEUDS_COTE+1);
        ++couleur.bordure;
        if(couleur.bordure == 2) g.removeArc(n+NB_NOEUDS_COTE, n+NB_NOEUDS_COTE+1);
        else g.coloreArc(n+NB_NOEUDS_COTE, n+NB_NOEUDS_COTE+1, couleur); }
    if(g.existeArc(n+1, n+NB_NOEUDS_COTE+1)) {
        couleur = g.couleurArc(n+1, n+NB_NOEUDS_COTE+1);
        ++couleur.bordure;
        if(couleur.bordure == 2) g.removeArc(n+1, n+NB_NOEUDS_COTE+1);
        else g.coloreArc(n+1, n+NB_NOEUDS_COTE+1, couleur); }
    if(g.existeArc(n+1, n)) {
        couleur = g.couleurArc(n+1, n);
        ++couleur.bordure;
        if(couleur.bordure == 2) g.removeArc(n+1, n);
        else g.coloreArc(n+1, n, couleur); }
    if(g.existeArc(n+NB_NOEUDS_COTE, n)) {
        couleur = g.couleurArc(n+NB_NOEUDS_COTE, n);
        ++couleur.bordure;
        if(couleur.bordure == 2) g.removeArc(n+NB_NOEUDS_COTE, n);
        else g.coloreArc(n+NB_NOEUDS_COTE, n, couleur); }
    if(g.existeArc(n+NB_NOEUDS_COTE+1, n+NB_NOEUDS_COTE)) {
        couleur = g.couleurArc(n+NB_NOEUDS_COTE+1, n+NB_NOEUDS_COTE);
        ++couleur.bordure;
        if(couleur.bordure == 2) g.removeArc(n+NB_NOEUDS_COTE+1, n+NB_NOEUDS_COTE);
        else g.coloreArc(n+NB_NOEUDS_COTE+1, n+NB_NOEUDS_COTE, couleur); }
    if(g.existeArc(n+NB_NOEUDS_COTE+1, n+1)) {
        couleur = g.couleurArc(n+NB_NOEUDS_COTE+1, n+1);
        ++couleur.bordure;
        if(couleur.bordure == 2) g.removeArc(n+NB_NOEUDS_COTE+1, n+1);
        else g.coloreArc(n+NB_NOEUDS_COTE+1, n+1, couleur); }
}
/* void makeTrous(Graphe<char, CA> &g, Chaine<int> &trous)
 * la case nord-ouest etant utilisee comme point d'origine pour tous les calculs, on s'interdit de la retirer
 * fait NB_CASES*PROPORTION_TROUS trous aleatoires dans g ou retire les cases qui sont dans trousImposes, puis retire toutes les cases qui ne sont plus accessibles depuis la case nord-ouest */
void makeTrous(Graphe<char, CA> &g, Chaine<int> &trous) {
    int i, n;
    CA couleur;
    bool isTrou[NB_CASES];

    for(i = 0; i < NB_CASES; ++i)
        isTrou[i] = false;
    srand((unsigned int)time(NULL));
    if(imposeTrous) {
        Chaine<unsigned int>::Iterator it(trousImposes, 0);
        while(!it.end()) {
            n = *it;
            if((!isTrou[(n/NB_NOEUDS_COTE)*(NB_NOEUDS_COTE-1) + n%NB_NOEUDS_COTE]) && (n != 0)) {
                trous.addElem(-1, n);
                ajouteTrou(g, n);
                isTrou[(n/NB_NOEUDS_COTE)*(NB_NOEUDS_COTE-1) + n%NB_NOEUDS_COTE] = true; }
            ++it; } }
    else {
        int des = 1 - (NB_NOEUDS_COTE%2);
        for(i = 0; i < NB_CASES*PROPORTION_TROUS; ++i) {
            n = (rand()%(NB_NOEUDS_COTE-1))*NB_NOEUDS_COTE + rand()%(NB_NOEUDS_COTE-1);
            if((!isTrou[n]) && (n != 0)) {
                trous.addElem(-1, n);
                ajouteTrou(g, n);
                isTrou[(n/NB_NOEUDS_COTE)*(NB_NOEUDS_COTE-1) + n%NB_NOEUDS_COTE] = true;
                des += 2*((n/NB_NOEUDS_COTE + n%NB_NOEUDS_COTE)%2)-1; } }
        while(des != 0) { //petite heuristique : s'il n'y a pas autant de cases noires que de cases blanches, on sait que le domaine n'est pas pavable
```

```
            n = (rand()%(NB_NOEUDS_COTE-1))*NB_NOEUDS_COTE + rand()%(NB_NOEUDS_COTE-1);
            if((!isTrou[n]) && (n != 0)) {
                if((des < 0) && ((n/NB_NOEUDS_COTE + n%NB_NOEUDS_COTE)%2 == 1)) {
                    trous.addElem(-1, n);
                    ajouteTrou(g, n);
                    isTrou[(n/NB_NOEUDS_COTE)*(NB_NOEUDS_COTE-1) + n%NB_NOEUDS_COTE] = true;
                    ++des; }
                if((0 < des) && ((n/NB_NOEUDS_COTE + n%NB_NOEUDS_COTE)%2 == 0)) {
                    trous.addElem(-1, n);
                    ajouteTrou(g, n);
                    isTrou[(n/NB_NOEUDS_COTE)*(NB_NOEUDS_COTE-1) + n%NB_NOEUDS_COTE] = true;
                    --des; } } } }
    // on enleve les cases qui ne sont pas accessibles depuis le sommet en haut à gauche du graphe
    Chaine<unsigned int> noeuds(1, 0), suivants;
    Chaine<unsigned int>::Iterator it;
    while(noeuds.taille() != 0) {
        n = noeuds[0];
        noeuds.supprElem(0);
        i = (n/NB_NOEUDS_COTE)*(NB_NOEUDS_COTE-1) + n%NB_NOEUDS_COTE;
        if((i < NB_CASES) && (n%NB_NOEUDS_COTE != NB_NOEUDS_COTE-1) && (!isTrou[i])) {
            isTrou[i] = true;
            suivants = g.suivants(n);
            it = suivants.begin();
            while(!it.end()) {
                noeuds.addElem(-1, *it);
                ++it; } } }
    for(i = 0; i < NB_CASES; ++i) {
        if(!isTrou[i]) {
            n = (i/(NB_NOEUDS_COTE-1))*NB_NOEUDS_COTE + i%(NB_NOEUDS_COTE-1);
            trous.addElem(-1, n);
            ajouteTrou(g, n); } }
}
/* void rayerTrousCarre(Window &fen, Chaine<int> const& trous) : grise les cases d'indice i pour i dans trous sur la fenetre fen */
void griserTrousCarre(Window &fen, Chaine<int> const& trous) {
    Chaine<int>::Iterator it(trous, 0);
    int lig, col;

    fen.setColor(128, 128, 128);
    while(!it.end()) {
        lig = (*it)/NB_NOEUDS_COTE;
        col = (*it)%NB_NOEUDS_COTE;
        fen.drawRect(1+T_CASE*col, 1+T_CASE*lig, T_CASE, T_CASE);
        ++it; }
}
/* int equilibreTrou(Graphe<char, CA> &g, Graphe<char, char> const& tetesTrous, int *valuations, int trou, UnionFind<NB_CASES> &noeuds) : calcule la valeur d'equilibre et affecte celle-ci aux aretes qui croisent la ligne allant du centre du trou trou et le centre du premier trou rencontre en allant vers le nord */
int equilibreTrou(Graphe<char, CA> &g, Graphe<char, char> const& tetesTrous, int *valuations, int trou, UnionFind<NB_CASES> &noeuds) {
    int eq = 0;
    Chaine<unsigned int> suivants = tetesTrous.suivants(trou);
    Chaine<unsigned int>::Iterator it(suivants, 0);

    while(!it.end()) {
        eq += equilibreTrou(g, tetesTrous, valuations, *it, noeuds);
        ++it; }
    if(trou != NB_CASES) {
        int i;
        eq += valuations[trou];
        do{
            i = trou + trou/(NB_NOEUDS_COTE-1);
            if(g.existeArc(i, i+1)) {
                CA couleur = g.couleurArc(i, i+1);
                couleur.equilibre = eq;
                g.coloreArc(i, i+1, couleur);
                couleur.spin = -couleur.spin;
                couleur.equilibre = -eq;
                g.coloreArc(i+1, i, couleur); }
            trou -= NB_NOEUDS_COTE-1;
        } while((0 <= trou) && (noeuds.find(trou) == -1)); }
```

```
    return eq;
}
/* void equilibrage(Window &fen, Graphe<char, CA> &g, Chaine<int> const& trous) : calcule pour chaque amas connexe de trous une case "parent" faisant partie de l'amas et appelle le calcul des equilibres */
void equilibrage(Window &fen, Graphe<char, CA> &g, Chaine<int> const& trous, UnionFind<NB_CASES> &noeuds) {
    int i;
    Graphe<char, char> tetesTrous(NB_CASES+1); //toutes les cases + l'extérieur
    Chaine<int>::Iterator it(trous, 0);

    noeuds.delie();
    // si i est le numéro d'un noeud, le numéro de la case correspondante est (i/NB_NOEUDS_COTE)*(NB_NOEUDS_COTE-1) + i%NB_NOEUDS_COTE
    while(!it.end()) {
        i = *it;
        i = (i/NB_NOEUDS_COTE)*(NB_NOEUDS_COTE-1) + i%NB_NOEUDS_COTE;
        ++it;
        noeuds.unit(i, i);
        if((i%(NB_NOEUDS_COTE-1) != 0) && (noeuds.find(i-1) != -1)) noeuds.unit(i, i-1); //W
            if((i/(NB_NOEUDS_COTE-1) != 0) && (noeuds.find(i-NB_NOEUDS_COTE+1) != -1)) noeuds.unit(i, i-NB_NOEUDS_COTE+1); } //N
    it = trous.begin();
    while(!it.end()) {
        i = *it;
        i = (i/NB_NOEUDS_COTE)*(NB_NOEUDS_COTE-1) + i%NB_NOEUDS_COTE;
        ++it;
        if(noeuds.find(i) == i){
            int j = i-NB_NOEUDS_COTE+1;
            while((0 <= j) && (noeuds.find(j) == -1)) j -= NB_NOEUDS_COTE-1;
            if(j < 0) tetesTrous.addArc(NB_CASES, i);
            else tetesTrous.addArc(noeuds.find(j), i);
            i += i/(NB_NOEUDS_COTE-1);
            j += j/(NB_NOEUDS_COTE-1);
            relierCaseCarree(fen, floor(float(i)/NB_NOEUDS_COTE), i%NB_NOEUDS_COTE, floor(float(j)/NB_NOEUDS_COTE), i%NB_NOEUDS_COTE); } }
    int valuations[NB_CASES] = {0}; // valuations[i] correspond a la valeur de l'equilibre pour i si la case i est le parent de l'amas de trous auquel il appartient et si cet amas etait seul dans le graphe
    for(i = 0; i < NB_CASES; ++i) {
        if(noeuds.find(i) != -1) {
            valuations[noeuds.find(i)] += 4 - 8*((i/(NB_NOEUDS_COTE-1) + i%(NB_NOEUDS_COTE-1))%2); } }
    equilibreTrou(g, tetesTrous, valuations, NB_CASES, noeuds);
}
/* void pavageCarreMin(Graphe<char, CA> const& g, Graphe<char, CA> const& arbre, int hauteur[NB_NOEUDS], Chaine<unsigned int> const& bordure) : calcul du pavage minimal de g dans le tableau hauteur, connaissant un arbre couvrant de g et la bordure de g */
void pavageCarreMin(Graphe<char, CA> const& g, Graphe<char, CA> const& arbre, int hauteur[NB_NOEUDS], int v0) {
    Chaine<unsigned int> noeuds, suivants;
    int min[NB_NOEUDS], max[NB_NOEUDS], i;
    Chaine<unsigned int>::Iterator it;
    CA couleur;

    for(i = 0; i < NB_NOEUDS; ++i) min[i] = max[i] = hauteur[i] = -(1 << 31); //-(1 << 31) = -infty : on n'aura pas de graphe assez grand pour que la hauteur d'un noeud puisse descendre a -(1 << 31)
    min[v0] = max[v0] = hauteur[v0] = 0;
    noeuds.addElem(-1, v0);
    //calcul de min[i] = hauteur[i] et max[i] pour tout noeud i en suivant le chemin entre 0 et i dans arbre
    while(noeuds.taille() != 0) {
        i = noeuds[0];
        noeuds.supprElem(0);
        suivants = arbre.suivants(i);
        it = suivants.begin();
        while(!it.end()) {
            if(min[*it] == -(1 << 31)) {
                couleur = g.couleurArc(i, *it);
                if(couleur.bordure == 1) {
                    min[*it] = hauteur[*it] = min[i] + couleur.equilibre + couleur.spin;
                    max[*it] = max[i] + couleur.equilibre + couleur.spin; }
```

```
                else {
                    min[*it] = hauteur[*it] = min[i] + couleur.equilibre - couleur.spin - 2;
                    max[*it] = max[i] + couleur.equilibre - couleur.spin + 2; }
                noeuds.addElem(-1, *it); }
            ++it; } }
    //on recupere les noeuds i tels qu'il existe i' adjacent a i tel que hauteur[i] + t(i, i') < hauteur[i']
    for(i = 0; i < NB_NOEUDS; ++i) {
        suivants = g.suivants(i);
        it = suivants.begin();
        while(!it.end()) {
            couleur = g.couleurArc(i, *it);
            if(((couleur.bordure == 1) && (hauteur[i] + couleur.equilibre + couleur.spin < hauteur[*it])) || (hauteur[i] + couleur.equilibre - couleur.spin + 2 < hauteur[*it])) {
                noeuds.addElem(-1, i);
                break; }
            ++it; } }
    //tant qu'il y a des noeuds verifiant la condition precedente, on augmente leur hauteur de 4
    while(noeuds.taille() != 0) {
        i = noeuds[0];
        hauteur[i] += 4;
        if(max[i] < hauteur[i]) {// probleme lors du calcul : on a depasse max[i], donc il n'y a pas de pavage de g
            hauteur[v0] = -2;
            return; }
        noeuds.supprElem(0);
        suivants = g.suivants(i);
        it = suivants.begin();
        bool aRemettre = false;
        while(!it.end()) {
            couleur = g.couleurArc(i, *it);
            if(((couleur.bordure == 1) && (hauteur[i] + couleur.equilibre + couleur.spin < hauteur[*it])) || (hauteur[i] + couleur.equilibre - couleur.spin + 2 < hauteur[*it])) aRemettre = true;
            if((((couleur.bordure == 1) && (hauteur[*it] - couleur.equilibre - couleur.spin < hauteur[i])) || (hauteur[*it] - couleur.equilibre + couleur.spin + 2 < hauteur[i])) && (noeuds.seek(*it) == -1)) noeuds.addElem(-1, *it);
            ++it; }
        if(aRemettre) noeuds.addElem(-1, i); }
}
/* void pavageCarreMax(Graphe<char, CA> const& g, Graphe<char, CA> const& arbre, int hauteur[NB_NOEUDS], Chaine<unsigned int> const& bordure) : calcul du pavage maximal de g */
void pavageCarreMax(Graphe<char, CA> const& g, Graphe<char, CA> const& arbre, int hauteur[NB_NOEUDS], int v0) {
    Chaine<unsigned int> noeuds, suivants;
    int min[NB_NOEUDS], max[NB_NOEUDS], i;
    Chaine<unsigned int>::Iterator it;
    CA couleur;

    for(i = 0; i < NB_NOEUDS; ++i) min[i] = max[i] = hauteur[i] = -(1 << 31);
    min[v0] = max[v0] = hauteur[v0] = 0;
    noeuds.addElem(-1, v0);
    //on pose ici hauteur[i] = max[i]
    while(noeuds.taille() != 0) {
        i = noeuds[0];
        noeuds.supprElem(0);
        suivants = arbre.suivants(i);
        it = suivants.begin();
        while(!it.end()) {
            if(min[*it] == -(1 << 31)) {
                couleur = g.couleurArc(i, *it);
                if(couleur.bordure == 1) {
                    min[*it] = min[i] + couleur.equilibre + couleur.spin;
                    max[*it] = hauteur[*it] = max[i] + couleur.equilibre + couleur.spin; }
                else {
                    min[*it] = min[i] + couleur.equilibre - couleur.spin - 2;
                    max[*it] = hauteur[*it] = max[i] + couleur.equilibre - couleur.spin + 2; }
                noeuds.addElem(-1, *it); }
            ++it; } }
    //on recupere les noeuds i tels qu'il existe i' adjacent a i tel que hauteur[i] < hauteur[i'] + b(i, i')
```

```
    for(i = 0; i < NB_NOEUDS; ++i) {
        suivants = g.suivants(i);
        it = suivants.begin();
        while(!it.end()) {
            couleur = g.couleurArc(i, *it);
            if(((couleur.bordure == 1) && (hauteur[*it] < hauteur[i] + couleur.equilibre + couleur.spin)) || (hauteur[*it] < hauteur[i] + couleur.equilibre - couleur.spin - 2)) {
                noeuds.addElem(-1, i);
                break; }
            ++it; } }
    while(noeuds.taille() != 0) {
        i = noeuds[0];
        max[i] = hauteur[i] -= 4;
        if(hauteur[i] < min[i]) {
            hauteur[v0] = -2;
            return; }
        noeuds.supprElem(0);
        suivants = g.suivants(i);
        it = suivants.begin();
        bool aRemettre = false;
        while(!it.end()) {
            couleur = g.couleurArc(i, *it);
            if(((couleur.bordure == 1) && (hauteur[*it] < hauteur[i] + couleur.equilibre + couleur.spin)) || (hauteur[*it] < hauteur[i] + couleur.equilibre - couleur.spin - 2)) aRemettre = true;
            if((((couleur.bordure == 1) && (hauteur[i] < hauteur[*it] - couleur.equilibre - couleur.spin)) || (hauteur[i] < hauteur[*it] - couleur.equilibre + couleur.spin - 2)) && (noeuds.seek(*it) == -1)) noeuds.addElem(-1, *it);
            ++it; }
        if(aRemettre) noeuds.addElem(-1, i); }
}
/* void placeTuile(Window &fen, unsigned int i, unsigned int j) : raye l'arete separant les cases i et j dans la FenGrapheetre FenGraphe */
void placeTuile(Window &fen, unsigned int i, unsigned int j) {
    if(j < i) placeTuile(fen, j, i);
    else {
        if(j-i == 1) fen.drawLine((i%NB_NOEUDS_COTE)*T_CASE + 1 + T_CASE/2, (i/NB_NOEUDS_COTE)*T_CASE + 1 - T_CASE/2, (i%NB_NOEUDS_COTE)*T_CASE + 1 + T_CASE/2, (i/NB_NOEUDS_COTE)*T_CASE + 1 + T_CASE/2);
        else fen.drawLine((i%NB_NOEUDS_COTE)*T_CASE + 1 - T_CASE/2, (i/NB_NOEUDS_COTE)*T_CASE + 1 + T_CASE/2, (i%NB_NOEUDS_COTE)*T_CASE + 1 + T_CASE/2, (i/NB_NOEUDS_COTE)*T_CASE + 1 + T_CASE/2); }
}
/* void dispPavage(Window &fen, Graphe<char, CA> const& g, int hauteur[NB_NOEUDS]) : trouve les aretes coupant une tuile dans le pavage decrit par hauteur : une arete coupe une tuile d'un pavage lorsque la difference de hauteur des noeuds aux extremites de l'arete n'est pas 1 ou -1 */
void dispPavage(Window &fen, Graphe<char, CA> const& g, int hauteur[NB_NOEUDS]) {
    unsigned int i;
    Chaine<unsigned int> suivants;
    Chaine<unsigned int>::Iterator it;

    fen.setColor(0, 255, 0);
    for(i = 0; i < NB_NOEUDS; ++i) {
        if(hauteur[i] != -1) {
            suivants = g.suivants(i);
            it = suivants.begin();
            while(!it.end()) {
                if(abs(hauteur[i]-hauteur[*it]+g.couleurArc(i, *it).equilibre) != 1)
                    placeTuile(fen, i, *it);
                ++it; } } }
}
/* void dispFractures(Window &fen, Graphe<char, CA> const& g, int hMin[NB_NOEUDS], int hMax[NB_NOEUDS]) : affiche les lignes de fractures de g : colore en rouge les aretes reliant deux noeuds qui ont la meme hauteur dans les pavages minimal et maximal de g */
void dispFractures(Window &fen, Graphe<char, CA> const& g, int hMin[NB_NOEUDS], int hMax[NB_NOEUDS]) {
    unsigned int i;
    Chaine<unsigned int> suivants;
    Chaine<unsigned int>::Iterator it;

    fen.setColor(255, 0, 0);
```

```
    for(i = 0; i < NB_NOEUDS; ++i) {
        suivants = g.suivants(i);
        it = suivants.begin();
        while(!it.end()) {
            if((hMin[i] == hMax[i]) && (hMin[*it] == hMax[*it]) && (abs(hMin[i]-hMin[*it]-g.couleurArc(*it, i).equilibre) == 1)) {
                fen.drawLine((i%NB_NOEUDS_COTE)*T_CASE + 1, (i/NB_NOEUDS_COTE)*T_CASE + 1, ((*it)%NB_NOEUDS_COTE)*T_CASE + 1, ((*it)/NB_NOEUDS_COTE)*T_CASE + 1);
                if(abs((int)i - (int)*it) == 1) {
                    fen.drawLine((i%NB_NOEUDS_COTE)*T_CASE + 1, (i/NB_NOEUDS_COTE)*T_CASE, ((*it)%NB_NOEUDS_COTE)*T_CASE + 1, ((*it)/NB_NOEUDS_COTE)*T_CASE);
                    fen.drawLine((i%NB_NOEUDS_COTE)*T_CASE + 1, (i/NB_NOEUDS_COTE)*T_CASE + 2, ((*it)%NB_NOEUDS_COTE)*T_CASE + 1, ((*it)/NB_NOEUDS_COTE)*T_CASE + 2); }
                else {
                    fen.drawLine((i%NB_NOEUDS_COTE)*T_CASE, (i/NB_NOEUDS_COTE)*T_CASE + 1, ((*it)%NB_NOEUDS_COTE)*T_CASE, ((*it)/NB_NOEUDS_COTE)*T_CASE + 1);
                    fen.drawLine((i%NB_NOEUDS_COTE)*T_CASE + 2, (i/NB_NOEUDS_COTE)*T_CASE + 1, ((*it)%NB_NOEUDS_COTE)*T_CASE + 2, ((*it)/NB_NOEUDS_COTE)*T_CASE + 1); } }
            ++it; } }
}
class FenGraphe : public Window {
    Graphe<char, CA> graphe, arbre;
    Chaine<int> trous;
    int etape, hMin[NB_NOEUDS], hMax[NB_NOEUDS], i;
    Screenshot *screen;
    FILE *fSave;
    UnionFind<NB_CASES> amasTrous;
public:
    FenGraphe(int h, int w) : Window(h, w), graphe(NB_NOEUDS), arbre(NB_NOEUDS), trous(), etape(0), screen(nullptr), fSave(nullptr), amasTrous() {}
    ~FenGraphe() { delete screen; }
    void sourisPressEvent(Event &boutons) {
        switch(etape) {
            case -1: //si g n'a pas de pavage, on reinitialise tout ici
                graphe.reinit();
                arbre.reinit();
                trous.vider();
                setColor(255, 255, 255);
                drawRect(0, 0, w(), h());
                ++etape;
            case 0: //on affiche la grille carree, on construit g et on le troue
                drawGrilleCarree(*this);
                makeGrapheCarre(graphe);
                makeTrous(graphe, trous);
                griserTrousCarre(*this, trous);
                update();
                break;
            case 1://equilibrage de g
                screen = new Screenshot(*this);
                treeSort(trous);
                equilibrage(*this, graphe, trous, amasTrous);
                amasTrous.unit(0, 0);
                i = 0;
                update();
                break;
            case 2://calcul et eventuel affichage du pavage minimal de g pour v0 = 0
                screen->afficher(*this);
                graphe.foretCouvrante(arbre);
                pavageCarreMin(graphe, arbre, hMin/*, bordure*/, 0);
                if(hMin[0] == -2) etape = -2;
                else dispPavage(*this, graphe, hMin);
                update();
                break;
            case 3://calcul et affichage du pavage maximal de g pour v0 = 0
                screen->afficher(*this);
                pavageCarreMax(graphe, arbre, hMax/*, bordure*/, 0);
                if(hMax[0] == -2) etape = 4;
                else dispPavage(*this, graphe, hMax);
                update();
                break;
            case 4://calcul et affichage des lignes de fracture de g puis sauvegarde des trous
```

```
                    et
                   screen->afficher(*this);
                   dispFractures(*this, graphe, hMin, hMax);
                   for(i = 1; i < NB_CASES; ++i) {
                       int j = (i/(NB_NOEUDS_COTE-1))*NB_NOEUDS_COTE + i%(NB_NOEUDS_COTE-1);
#ifdef CONJ1
                       if(amasTrous.find(i) == i)
#else
                       if(arbre.suivants(j).taille() != 0) //on ne travaille que sur les noeuds
                           qui sont accessibles depuis le noeud 0
#endif
                           {
                               //bordure = bordureGraphe(graphe, trous, j);
                               pavageCarreMin(graphe, arbre, hMin/*, bordure*/, j);
                               if(hMin[j] == -2) continue;
                               pavageCarreMax(graphe, arbre, hMax/*, bordure*/, j);
                               if(hMax[j] == -2) continue;
                               dispFractures(*this, graphe, hMin, hMax);} }
                   update(); }
             ++etape; }
};
int main(int argc, char *argv[]) {
    App app(APP_VIDEO);
    FenGraphe fen(T_FEN, T_FEN);
    fen.show();
    return app.executer();
}
```

B.2 Application.h

```
#ifndef APP_H_INCLUDED
#define APP_H_INCLUDED
#include <SDL2/SDL.h>
#include "Chaine.h"
#define APP_TIMER SDL_INIT_TIMER
#define APP_AUDIO SDL_INIT_AUDIO
#define APP_VIDEO SDL_INIT_VIDEO
#define APP_JOYSTICK SDL_INIT_JOYSTICK
#define APP_HAPTIC SDL_INIT_HAPTIC
#define APP_GAME_CONTROLLER SDL_INIT_GAMECONTROLLER
#define APP_EVENTS SDL_INIT_EVENTS
#define APP_ALL SDL_INIT_EVERYTHING
#define APP_NO_PARACHUTE SDL_INIT_NOPARACHUTE
class Window;
typedef struct Event {
    bool toucheEnfoncee[255];
    bool boutonSourisEnfonce[8];
    int sourisX, sourisY, sourisXRelatif, sourisYRelatif;
    uint8_t bouton; } Event;
const char* lastError(){ return SDL_GetError(); }
class App{
    friend class Window;
public:
    App(uint32_t flags);
    ~App() { SDL_Quit(); }
    int executer();
private:
    SDL_Event _lastEvent;
    Event _boutons;
    Chaine<Window*> _fens; };
App *app = nullptr;
#include "Window.h"
App::App(uint32_t flags) : _lastEvent(), _boutons(), _fens(){
    if(app != nullptr) throw ("Une seule application a la fois !\n");
    if(SDL_Init(flags | SDL_INIT_VIDEO) != 0) throw SDL_GetError();
    app = this;
    return; }
static bool sameWindows(Window* const& fen, Uint32 const& windowID) { return windowID ==
    SDL_GetWindowID(fen->_fen); }
static void gainFocus(Uint32 const& windowID, Chaine<Window*> &fens) {
    int idWin = fens.seek(sameWindows, windowID);
    if(0 < idWin) {
        Window *win = fens[idWin];
```

```
        fens.supprElem(idWin);
        fens.addElem(-1, win); } }
int App::executer(){
    while(1){
        if(SDL_WaitEvent(&_lastEvent) != 1) return -1;
        do{
            if(_fens.taille() == 0) return 0;
            switch(_lastEvent.type){
            case SDL_QUIT: break;
            case SDL_KEYDOWN:
                _boutons.toucheEnfoncee[_lastEvent.key.keysym.scancode] = true;
                _boutons.bouton = _lastEvent.key.keysym.sym;
                _fens[0]->clavierPressEvent(_boutons);
                break;
            case SDL_KEYUP:
                _boutons.toucheEnfoncee[_lastEvent.key.keysym.scancode] = false;
                _boutons.bouton = _lastEvent.key.keysym.sym;
                _fens[0]->clavierReleaseEvent(_boutons);
                break;
            case SDL_MOUSEMOTION:
                _boutons.sourisX = _lastEvent.motion.x;
                _boutons.sourisY = _lastEvent.motion.y;
                _boutons.sourisXRelatif = _lastEvent.motion.xrel;
                _boutons.sourisYRelatif = _lastEvent.motion.yrel;
                _boutons.bouton = 0;
                break;
            case SDL_MOUSEBUTTONDOWN:
                _boutons.boutonSourisEnfonce[_lastEvent.button.button] = true;
                _boutons.bouton = _lastEvent.button.button;
                _fens[0]->sourisPressEvent(_boutons);
                break;
            case SDL_MOUSEBUTTONUP:
                _boutons.boutonSourisEnfonce[_lastEvent.button.button] = false;
                _boutons.bouton = _lastEvent.button.button;
                _fens[0]->sourisReleaseEvent(_boutons);
                break;
            case SDL_WINDOWEVENT:
                gainFocus(_lastEvent.window.windowID, _fens);
                _boutons.bouton = _lastEvent.window.event;
                if(_boutons.bouton == SDL_WINDOWEVENT_CLOSE) _fens[0]->~Window();
                else _fens[0]->windowEvent(_boutons); }
        }while(SDL_PollEvent(&_lastEvent)); }
    return 0; }
#endif // APP_H_INCLUDED
```

B.3 `Window.h`

```
#ifndef WINDOW_H_INCLUDED
#define WINDOW_H_INCLUDED
#include <SDL2/SDL.h>
#define WINPOS_UNDEF SDL_WINDOWPOS_UNDEFINED
#define WINDPOS_CENTRE SDL_WINDOWPOS_CENTERED
#define WIN_FULLSCREEN SDL_WINDOW_FULLSCREEN
#define WIN_FULLSCREEN_DESKTOP SDL_WINDOW_FULLSCREEN_DESKTOP
#define WIN_BORDERLESS SDL_WINDOW_BORDERLESS
#define WIN_RESIZABLE SDL_WINDOW_RESIZABLE
#define WIN_MINIMIZED SDL_WINDOW_MINIMIZED
#define WIN_MAXIMIZED SDL_WINDOW_MAXIMIZED
#define WIN_INPUT_GRABBED SDL_WINDOW_INPUT_GRABBED
#define WIN_HIGHDPI SDL_WINDOW_ALLOW_HIGHDPI
#ifndef MAX
 #define MAX(a, b) (a < b ? b : a)
#endif //MAX
class Img;
class Event;
class Window {
public:
    Window(int w = 640, int h = 480, int x = WINPOS_UNDEF, int y = WINPOS_UNDEF, const char* titre = "fenetre", uint32_t flags = 0);
    ~Window();
    int show();
    void hide();
    int w() const;
```

```
    int h() const;
    int drawLine(int xd, int yd, int xf, int yf);
    int drawRect(int x, int y, int w, int h);
    int drawImg(Img const& i, int x = 0, int y = 0);
    int setColor(unsigned int r, unsigned int v, unsigned int b, unsigned int a = 255);
    int update();
    virtual void sourisPressEvent(Event &boutons) {}
    virtual void sourisReleaseEvent(Event &boutons) {}
    virtual void clavierPressEvent(Event &boutons) {}
    virtual void clavierReleaseEvent(Event &boutons) {}
    virtual void windowEvent(Event &boutons) {}
    friend class Crayon;
    friend class Screenshot;
public:
    SDL_Window *_fen;
    SDL_Renderer *_renderer;
    SDL_Texture *_t; }; //on ecrit sur une texture pour ne pas repartir d'un ecran noir a chaque fois
#include "Application.h"
Window::Window(int w, int h, int x, int y, const char* titre, uint32_t flags) : _fen(NULL), _renderer(NULL) {
    if(app == NULL) throw "Pas de fenetre sans application !\n";
    if((_fen = SDL_CreateWindow(titre, MAX(x, 0), MAX(y, 0), MAX(w, 120), MAX(h, 10), flags | SDL_WINDOW_HIDDEN)) == NULL) goto endWindow;
    if((_renderer = SDL_CreateRenderer(_fen, -1, 0)) == NULL) goto endRenderer;
    if((_t = SDL_CreateTexture(_renderer, SDL_PIXELFORMAT_RGBA8888, SDL_TEXTUREACCESS_TARGET, w, h)) == NULL) goto endTexture;
    if(SDL_SetRenderTarget(_renderer, _t) < 0) goto endTarget;
    return;
endTarget: SDL_DestroyTexture(_t);
endTexture: SDL_DestroyRenderer(_renderer);
endRenderer: SDL_DestroyWindow(_fen);
endWindow: throw SDL_GetError(); }
Window::~Window() {
    if(_fen != NULL) {
        app->_fens.supprElem(app->_fens.seek(this));
        SDL_DestroyWindow(_fen);
        SDL_DestroyRenderer(_renderer);
        SDL_DestroyTexture(_t);
        _fen = nullptr; } }
int Window::show() { SDL_ShowWindow(_fen); app->_fens.addElem(-1, this); return 0; }
int Window::w() const { int w; SDL_GetWindowSize(_fen, &w, NULL); return w; }
int Window::h() const { int h; SDL_GetWindowSize(_fen, NULL, &h); return h; }
int Window::drawLine(int xd, int yd, int xf, int yf) { if(SDL_RenderDrawLine(_renderer, xd, yd, xf, yf) == -1) return -1; return 0; }
int Window::drawRect(int x, int y, int w, int h) {
    SDL_Rect rect;
    rect.x = x; rect.y = y; rect.w = w; rect.h = h;
    if(SDL_RenderFillRect(_renderer, &rect) == -1) return -1;
    return 0; }
int Window::setColor(unsigned int r, unsigned int v, unsigned int b, unsigned int a) {
    if(SDL_SetRenderDrawColor(_renderer, r, v, b, a) != 0) return -1;
    return 0; }
int Window::update() {
    if(SDL_SetRenderTarget(_renderer, NULL) < 0) return -1;
    SDL_RenderCopy(_renderer, _t, NULL, NULL);
    SDL_RenderPresent(_renderer);
    SDL_SetRenderTarget(_renderer, _t);
    return 0; }
#endif // WINDOW_H_INCLUDED
```

B.4 Screenshot.h

```
#ifndef Screenshot_h
#define Screenshot_h
#include "Window.h"
class Screenshot{
private: SDL_Texture *_texture;
    public:
    Screenshot(Window const& fen);
    ~Screenshot() { SDL_DestroyTexture(_texture); }
    void afficher(Window &fen) { SDL_RenderCopy(fen._renderer, _texture, NULL, NULL); } };
Screenshot::Screenshot(Window const& fen) {
```

```
    _texture = SDL_CreateTexture(fen._renderer, SDL_PIXELFORMAT_RGBA8888,
        SDL_TEXTUREACCESS_TARGET, fen.w(), fen.h());
    SDL_RenderPresent(fen._renderer);
    SDL_SetRenderTarget(fen._renderer, _texture);
    SDL_RenderCopy(fen._renderer, fen._t, NULL, NULL);
    SDL_RenderPresent(fen._renderer);
    SDL_SetRenderTarget(fen._renderer, fen._t); }
#endif /* Screenshot_h */
```

B.5 Graphe.h

```
#ifndef Graphe_h
#define Graphe_h
#include <iostream>
#include <stdlib.h>
#include <assert.h>
#include "Chaine.h"
#include "ArbreAVL.h"
template<typename CN, typename CA>
class Graphe {
private: struct Arete;
public:
    Graphe(size_t taille);
    ~Graphe();
    void reinit();
    void addArc(unsigned long depart, unsigned long arrivee);
    void removeArc(unsigned long depart, unsigned long arrivee);
    bool existeArc(unsigned long depart, unsigned long arrivee) const;
    void coloreArc(unsigned int depart, unsigned int arrivee, CA const& couleur);
    CA couleurArc(unsigned int depart, unsigned int arrivee) const;
    Chaine<unsigned int> suivants(unsigned long noeud) const;
    void foretCouvrante(Graphe<CN, CA> &foret) const;
protected:
    CN *_couleurs;
    ArbreAVL<Arete> **_listeAdjacence;
    size_t _taille; };
template<typename CN, typename CA>
struct Graphe<CN, CA>::Arete {
    void *n;
    CA couleur;
    Arete(void *noeud, CA couleur = CA()) : n(noeud), couleur(couleur) {}
    bool operator<(Arete const& a) { return n < a.n; }
    bool operator !=(Arete const& a) { return n != a.n; } };
template<typename CN, typename CA>
Graphe<CN, CA>::Graphe(size_t taille) : _listeAdjacence(), _taille(taille) {
    _couleurs = (CN*) malloc(taille*sizeof(CN));
    _listeAdjacence = (ArbreAVL<Arete>**) malloc(sizeof(ArbreAVL<Arete>*)*taille);
    for(int i = 0; i < taille; ++i) _listeAdjacence[i] = new ArbreAVL<Arete>(); }
template<typename CN, typename CA>
Graphe<CN, CA>::~Graphe() {
    for(int i = 0; i < _taille; ++i) delete _listeAdjacence[i];
    free(_listeAdjacence); free(_couleurs); }
template<typename CN, typename CA>
void Graphe<CN, CA>::reinit() { for(int i = 0; i < _taille; ++i) _listeAdjacence[i]->vider();
    }
template<typename CN, typename CA>
void Graphe<CN, CA>::addArc(unsigned long depart, unsigned long arrivee) { if((MAX(depart,
    arrivee) < _taille) && !(_listeAdjacence[depart]->contient(_listeAdjacence+arrivee)))
    _listeAdjacence[depart]->addElem(Arete(_listeAdjacence+arrivee)); }
template<typename CN, typename CA>
void Graphe<CN, CA>::removeArc(unsigned long depart, unsigned long arrivee) { if(MAX(depart,
    arrivee) < _taille) _listeAdjacence[depart]->supprElem(_listeAdjacence+arrivee); }
template<typename CN, typename CA>
bool Graphe<CN, CA>::existeArc(unsigned long depart, unsigned long arrivee) const { return
    _listeAdjacence[depart]->contient(_listeAdjacence+arrivee); }
template<typename CN, typename CA>
void Graphe<CN, CA>::coloreArc(unsigned int depart, unsigned int arrivee, CA const& couleur) {
    Arete &a = _listeAdjacence[depart]->seek(_listeAdjacence+arrivee);
    if(a.n == _listeAdjacence+arrivee) a.couleur = couleur; }
template<typename CN, typename CA>
CA Graphe<CN, CA>::couleurArc(unsigned int depart, unsigned int arrivee) const { return
    _listeAdjacence[depart]->seek(_listeAdjacence+arrivee).couleur; }
template<typename CN, typename CA>
```

```cpp
Chaine<unsigned int> Graphe<CN, CA>::suivants(unsigned long noeud) const {
    Chaine<unsigned int> result;
    if(noeud < _taille) {
        typename ArbreAVL<Graphe<CN, CA>::Arete>::Iterator it = _listeAdjacence[noeud]->begin();
        while(!it.end()) {
            result.addElem(-1, (unsigned int)( ((ArbreAVL<Graphe<CN, CA>::Arete>**)((*it).n)) - _listeAdjacence));
            ++it; }
        result.reverse(); }
    return result; }
template<typename CN, typename CA>
void Graphe<CN, CA>::foretCouvrante(Graphe<CN, CA> &foret) const {
    assert(foret._taille == _taille);
    int i, s, s2;
    char isVu[_taille];
    Chaine<unsigned int> sommets, adj;
    typename Chaine<unsigned int>::Iterator it;
    for(i = 0; i < _taille; ++i) isVu[i] = 0;
    i = 0;
    do{
        sommets.addElem(-1, i);
        while(sommets.taille() != 0) {
            s = sommets[0];
            sommets.supprElem(0);
            if(isVu[s] != 2) {
                isVu[s] = 2;
                adj = suivants(s);
                it = adj.begin();
                while(!it.end()) {
                    if(isVu[*it] == 0) {
                        s2 = *it;
                        foret.addArc(s, s2);
                        if(existeArc(s2, s)) foret.addArc(s2, s);
                        sommets.addElem(-1, s2);
                        isVu[s2] = 1; }
                    ++it; } } }
        while((isVu[i]) && (i < _taille)) ++i;
    }while(i < _taille); }
#endif /* Graphe_h */
```

B.6 ArbreAVL.h

```cpp
#ifndef ABREAVL_H
#define ABREAVL_H
template<typename T>
struct NoeudAVL {
    NoeudAVL *parent, *filsD, *filsG;
    char coefAVL; // profondeurD-profondeurG
    T val;
    bool gauche;
    NoeudAVL() : parent(nullptr), filsD(nullptr), filsG(nullptr), coefAVL(0), val(), gauche(false) {}
    NoeudAVL(T const& val) : parent(nullptr), filsD(nullptr), filsG(nullptr), coefAVL(0), val(val), gauche(false) {}
    NoeudAVL(NoeudAVL<T> const& n) : parent(n.parent), filsD(n.filsD), filsG(n.filsG), coefAVL(n.coefAVL), val(n.val), gauche(n.gauche) {}
    ~NoeudAVL(); };
template<typename T>
class ArbreAVL {
public:
    class Iterator;
    ArbreAVL() : _racine(nullptr) {}
    ~ArbreAVL();
    ArbreAVL<T>& addElem(T const& val);
    ArbreAVL<T>& supprElem(T const& val);
    void vider();
    template<typename T2>
    T& seek(T2 const& cle) const { return find(cle)->val; }
    template<typename T2>
    bool contient(T2 const& val) const { return find(val) != nullptr; }
    Iterator begin() const { return Iterator(*this); }
private:
```

```
    NoeudAVL<T> *_racine;
    template<typename T2>
    NoeudAVL<T>* find(T2 const& cle) const;
    void equilibrageAdd(NoeudAVL<T> *n);
    void equilibrageSuppr(NoeudAVL<T> *n);
    void rotationSG(NoeudAVL<T> *n);
    void rotationSD(NoeudAVL<T> *n);
    void rotationDG(NoeudAVL<T> *n);
    void rotationDD(NoeudAVL<T> *n); };
template<typename T>
class ArbreAVL<T>::Iterator {
public:
    Iterator() : _pointe(nullptr) {}
    Iterator(ArbreAVL<T> const& a);
    ArbreAVL<T>::Iterator& operator=(ArbreAVL<T>::Iterator const& i);
    ArbreAVL<T>::Iterator& operator++();
    bool end() const { return this->_pointe == nullptr; }
    T& operator*() const { return this->_pointe->val; }
private: NoeudAVL<T> *_pointe; };
template<typename T>
NoeudAVL<T>::~NoeudAVL() {
    if(this->filsG != nullptr) delete this->filsG;
    if(this->filsD != nullptr) delete this->filsD; }
template<typename T>
ArbreAVL<T>::~ArbreAVL() { delete this->_racine; this->_racine = nullptr; }
template<typename T>
ArbreAVL<T>& ArbreAVL<T>::addElem(T const& val) {
    NoeudAVL<T> *parent = this->_racine, *nouveau = new NoeudAVL<T>(val);
    bool insertion = true;
    if(this->_racine == nullptr) { this->_racine = nouveau; insertion = false; }
    while(insertion) {
        if(nouveau->val < parent->val) {
            if(parent->filsG == nullptr) {
                parent->filsG = nouveau;
                nouveau->parent = parent;
                nouveau->gauche = true;

                this->equilibrageAdd(nouveau);
                insertion = false; }
            else parent = parent->filsG; } // parent->filsG != nullptr
        else { // parent->val <= nouveau->val
            if(parent->filsD == nullptr) {
                parent->filsD = nouveau;
                nouveau->parent = parent;
                this->equilibrageAdd(nouveau);
                insertion = false; }
            else parent = parent->filsD; } } // parent->filsD != nullptr
    return *this; }
template<typename T>
ArbreAVL<T>& ArbreAVL<T>::supprElem(T const& val) {
    NoeudAVL<T> *aSuppr = this->find(val);
    if(aSuppr->filsG == nullptr) {
        if(aSuppr->filsD == nullptr) {
            if(aSuppr->parent == nullptr) this->_racine = nullptr;
            else { // aSuppr->parent != nullptr
                if(aSuppr->gauche) aSuppr->parent->filsG = nullptr;
                else aSuppr->parent->filsD = nullptr; // !aSuppr->gauche
                this->equilibrageSuppr(aSuppr); } }
        else { // aSuppr->filsD != nullptr
            if(aSuppr->parent == nullptr) {
                this->_racine = aSuppr->filsD;
                aSuppr->filsD->parent = nullptr; }
            else { // aSuppr->parent != nullptr
                aSuppr->filsD->parent = aSuppr->parent;
                if(aSuppr->gauche) {
                    aSuppr->filsD->gauche = true;
                    aSuppr->parent->filsG = aSuppr->filsD; }
                else aSuppr->parent->filsD = aSuppr->filsD; // !aSuppr->gauche
                this->equilibrageSuppr(aSuppr); } } }
    else { // aSuppr->filsG != nullptr
        if(aSuppr->filsD == nullptr) {
            if(aSuppr->parent == nullptr) {
                this->_racine = aSuppr->filsG;
```

```cpp
                aSuppr->filsG->parent = nullptr; }
            else { // aSuppr->parent != nullptr
                aSuppr->filsG->parent = aSuppr->parent;
                if(aSuppr->gauche) aSuppr->parent->filsG = aSuppr->filsG;
                else { // !aSuppr->gauche
                    aSuppr->filsG->gauche = false;
                    aSuppr->parent->filsD = aSuppr->filsG; }
                this->equilibrageSuppr(aSuppr); } }
        else { // aSuppr->filsD != nullptr
            NoeudAVL<T> *ssMax = aSuppr->filsG, *aEquilibrer;
            bool ssMaxGauche;
            while(ssMax->filsD != nullptr) ssMax = ssMax->filsD;
            ssMaxGauche = ssMax->gauche;
            if(ssMax->gauche) aEquilibrer = ssMax;
            else { // !ssMax->gauche
                aEquilibrer = ssMax->parent;
                if(ssMax->filsG != nullptr) {
                    ssMax->filsG->parent = ssMax->parent;
                    ssMax->filsG->gauche = false; }
                ssMax->parent->filsD = ssMax->filsG;
                ssMax->filsG = aSuppr->filsG;
                aSuppr->filsG->parent = ssMax; }
            ssMax->coefAVL = aSuppr->coefAVL;
            ssMax->gauche = aSuppr->gauche;
            ssMax->parent = aSuppr->parent;
            ssMax->filsD = aSuppr->filsD;
            if(aSuppr->parent == nullptr) this->_racine = ssMax;
            else { // aSuppr->parent != nullptr
                if(aSuppr->gauche) aSuppr->parent->filsG = ssMax;
                else aSuppr->parent->filsD = ssMax; } // !aSuppr->gauche
            aSuppr->filsD->parent = ssMax;
            aSuppr->parent = aEquilibrer;
            aSuppr->gauche = ssMaxGauche;
            this->equilibrageSuppr(aSuppr); } }
    aSuppr->filsD = aSuppr->filsG = nullptr;
    delete aSuppr; return *this; }
template<typename T>
void ArbreAVL<T>::vider() { delete this->_racine; this->_racine = nullptr; }
template<typename T>
void ArbreAVL<T>::equilibrageAdd(NoeudAVL<T> *n) {
    bool recure = true;
    do{ if(n->parent != nullptr) {
            if(n->gauche) {
                --n->parent->coefAVL;
                if(n->parent->coefAVL != -1) recure = false; }
            else {
                ++n->parent->coefAVL;
                if(n->parent->coefAVL != 1) recure = false; }

            if(n->parent->coefAVL < -1) {
                if(n->parent->filsG->coefAVL == 1) this->rotationDD(n->parent);
                else this->rotationSD(n->parent); }
            else if(n->parent->coefAVL > 1) {
                if(n->parent->filsD->coefAVL == -1) this->rotationDG(n->parent);
                else this->rotationSG(n->parent); }
            n = n->parent; }
        else recure = false; }while(recure); return; }
template<typename T>
void ArbreAVL<T>::equilibrageSuppr(NoeudAVL<T> *n) {
    bool recure = true;
    do{ if(n->parent == nullptr) recure = false;
        else {
            if(n->gauche) {
                ++n->parent->coefAVL;
                if(n->parent->coefAVL == 1) recure = false; }
            else {
                --n->parent->coefAVL;
                if(n->parent->coefAVL == -1) recure = false; }
            if(n->parent->coefAVL < -1) {
                n = n->parent;
                if(n->filsG->coefAVL == 0) recure = false;
                if(n->filsG->coefAVL == 1) this->rotationDD(n);
                else this->rotationSD(n); }
```

```
            else if(n->parent->coefAVL > 1) {
                n = n->parent;
                if(n->filsD->coefAVL == 0) recure = false;
                if(n->filsD->coefAVL == -1) this->rotationDG(n);
                else this->rotationSG(n); } }
        n = n->parent; }while(recure); }
template<typename T>
void ArbreAVL<T>::rotationSG(NoeudAVL<T> *n) {
    //mise a jour des coefsAVL
    n->coefAVL = 1-n->filsD->coefAVL;
    if(n->filsD->coefAVL == 0) n->filsD->coefAVL = -1;
    else n->filsD->coefAVL = 0;
    //mise a jour du parent du noeud principal
    if(n == this->_racine) this->_racine = n->filsD;
    else {
        if(n->gauche) n->parent->filsG = n->filsD;
        else n->parent->filsD = n->filsD; }
    //mise a jour des gauche
    n->filsD->gauche = n->gauche;
    n->gauche = true;
    if(n->filsD->filsG != nullptr) {
        n->filsD->filsG->gauche = false;
        n->filsD->filsG->parent = n; }
    //deplacement des noeuds
    n->filsD->parent = n->parent;
    n->parent = n->filsD;
    n->filsD = n->filsD->filsG;
    n->parent->filsG = n; }
template<typename T>
void ArbreAVL<T>::rotationSD(NoeudAVL<T> *n) {
    //mise a jour des coefsAVL
    n->coefAVL = -1-n->filsG->coefAVL;
    if(n->filsG->coefAVL == 0) n->filsG->coefAVL = -n->coefAVL;
    else n->filsG->coefAVL = 0;
    //mise a jour du parent du noeud principal
    if(n == this->_racine) this->_racine = n->filsG;
    else{
        if(n->gauche) n->parent->filsG = n->filsG;
        else n->parent->filsD = n->filsG; }
    //mise a jour des gauche
    n->filsG->gauche = n->gauche;
    n->gauche = false;
    if(n->filsG->filsD != nullptr) {
        n->filsG->filsD->gauche = true;
        n->filsG->filsD->parent = n; }
    //deplacement des noeuds
    n->filsG->parent = n->parent;
    n->parent = n->filsG;
    n->filsG = n->filsG->filsD;
    n->parent->filsD = n;}
template<typename T>
void ArbreAVL<T>::rotationDG(NoeudAVL<T> *n) {
    //mise a jour des coefsAVL
    if(n->filsD->filsG->coefAVL == 1) {
        n->coefAVL = -1;
        n->filsD->coefAVL = 0; }
    else {
        n->coefAVL = 0;
        n->filsD->coefAVL = -n->filsD->filsG->coefAVL; }
    n->filsD->filsG->coefAVL = 0;
    //mise a jour du parent du noeud principal
    if(n == this->_racine) this->_racine = n->filsD->filsG;
    else {
        if(n->gauche) n->parent->filsG = n->filsD->filsG;
        else n->parent->filsD = n->filsD->filsG; }
    //mise a jour des gauche
    n->filsD->filsG->gauche = n->gauche;
    n->gauche = true;
    if(n->filsD->filsG->filsG != nullptr) {
        n->filsD->filsG->filsG->gauche = false;
        n->filsD->filsG->filsG->parent = n; }
    if(n->filsD->filsG->filsD != nullptr) {
        n->filsD->filsG->filsD->gauche = true;
```

```
        n->filsD->filsG->filsD->parent = n->filsD; }
    //deplacement des noeuds
    n->filsD->filsG->parent = n->parent;
    n->filsD->parent = n->parent = n->filsD->filsG;
    n->filsD->filsG = n->parent->filsD;
    n->parent->filsD = n->filsD;
    n->filsD = n->parent->filsG;
    n->parent->filsG = n;}
template<typename T>
void ArbreAVL<T>::rotationDD(NoeudAVL<T> *n) {
    //mise a jour des coefsAVL
    if(n->filsG->filsD->coefAVL == 1) {
        n->coefAVL = 0;
        n->filsG->coefAVL = -1; }
    else {
        n->coefAVL = -n->filsG->filsD->coefAVL;
        n->filsG->coefAVL = 0; }
    n->filsG->filsD->coefAVL = 0;
    //mise a jour du parent du noeud principal
    if(n == this->_racine) this->_racine = n->filsG->filsD;
    else {
        if(n->gauche) n->parent->filsG = n->filsG->filsD;
        else n->parent->filsD = n->filsG->filsD; }
    //mise a jour des gauche
    n->filsG->filsD->gauche = n->gauche;
    n->gauche = false;
    if(n->filsG->filsD->filsG != nullptr) {
        n->filsG->filsD->filsG->gauche = false;
        n->filsG->filsD->filsG->parent = n->filsG; }
    if(n->filsG->filsD->filsD != nullptr) {
        n->filsG->filsD->filsD->gauche = true;
        n->filsG->filsD->filsD->parent = n; }
    //deplacement des noeuds
    n->filsG->filsD->parent = n->parent;
    n->parent = n->filsG->filsD;
    n->filsG->parent = n->parent;
    n->filsG->filsD = n->parent->filsG;
    n->parent->filsG = n->filsG;
    n->filsG = n->parent->filsD;
    n->parent->filsD = n; }
template<typename T>
template<typename T2>
NoeudAVL<T>* ArbreAVL<T>::find(T2 const& cle) const {
    NoeudAVL<T> *actuel = this->_racine;
    while((actuel != nullptr) && (actuel->val != cle)) {
        if(actuel->val < cle) actuel = actuel->filsD;
        else actuel = actuel->filsG; }
    return actuel; }
template<typename T>
ArbreAVL<T>::Iterator::Iterator(ArbreAVL<T> const& a) : _pointe(a._racine) { if(this->_pointe != nullptr) while(this->_pointe->filsG != nullptr) this->_pointe = this->_pointe->filsG; }
template<typename T>
typename ArbreAVL<T>::Iterator& ArbreAVL<T>::Iterator::operator=(ArbreAVL<T>::Iterator const& i) { this->_pointe = i._pointe; return *this; }
template<typename T>
typename ArbreAVL<T>::Iterator& ArbreAVL<T>::Iterator::operator++() {
    if(this->_pointe != nullptr) {
        if(this->_pointe->filsD != nullptr) {
            this->_pointe = this->_pointe->filsD;
            while(this->_pointe->filsG != nullptr) this->_pointe = this->_pointe->filsG; }
        else {
            while((this->_pointe != nullptr) && (!this->_pointe->gauche)) this->_pointe = this->_pointe->parent;
            if(this->_pointe != nullptr) this->_pointe = this->_pointe->parent; } }
    return *this; }
#endif /* ArbreAVL_h */
```

B.7 `Chaine.h`

```
#ifndef CHAINE_H_INCLUDED
#define CHAINE_H_INCLUDED
#include <cstdio>
#include <cmath>
```

```cpp
#include <cstring>
#include <stdexcept>
#include <cstdint>
#include <initializer_list>
#include <functional>
template<typename T>
class Chaine {
protected:
    struct Element;
    Element *_debut;
    unsigned int _taille;
    public:
    class Iterator;
Chaine() : _debut((Chaine<T>::Element*)0), _taille(0) {}
    Chaine(int nbElements);
    Chaine(int nbElements, T const& val);
    Chaine(Chaine<T> const& c); /** ... de copie*/
    Chaine(std::initializer_list<T> const& init);
    ~Chaine() { this->vider(); }
    void addElem(int idPrecedent, T const& val);
    void supprElem(int id);
    void vider();
    void reverse();
    Chaine<T>& operator=(Chaine<T> const& c);
    T& operator[](int id) const;
    unsigned int taille() const { return this->_taille; }
    virtual int seek(T const& val) const;
    virtual int seek(Chaine<T> const& c) const;
    template<typename T2>
    int seek(bool (*filtre) (T const&, T2 const&), T2 const& cle);
    Chaine<T>::Iterator begin() const { return Chaine<T>::Iterator(*this, 0); } };
template<typename T>
struct Chaine<T>::Element {
    T val;
    Element *suivant;
    Element() : val(), suivant(nullptr) {}
    Element(T const& val, Element *suivant = nullptr) : val(val), suivant(suivant) {} };
template<typename T>
class Chaine<T>::Iterator {
protected: Element *_pointe;
public:
    Iterator() : _pointe(nullptr) {}
    Iterator(Chaine<T> const& c, int id);
    T& operator*() const { return this->_pointe->val; }
    Iterator& operator++();
    bool end() { return this->_pointe == nullptr; } };
template<typename T>
Chaine<T>::Chaine(int nbElements) : _debut((Chaine<T>::Element*)0), _taille(0) {
    if(nbElements <= 0) return ; /** plus rapide que de crÈer le premier ÈlÈment pour le
        dÈaruire ensuite */
    Chaine<T>::Element *actuel;
    this->_taille = nbElements;
/** crÈation du premier ÈlÈment */
    this->_debut = new Chaine<T>::Element();
    --nbElements;
    actuel = this->_debut;
/** crÈation des ÈlÈments suivant */
    while(nbElements > 0) {
        actuel->suivant = new Chaine<T>::Element();
        actuel = actuel->suivant;
        --nbElements; }
    actuel->suivant = (Chaine<T>::Element*) 0; /** le dernier ÈlÈment n'a pas de suivant */
    return; }
template<typename T>
Chaine<T>::Chaine(int nbElements, T const& val) : _debut((Chaine<T>::Element*)0), _taille(0) {
    if(nbElements <= 0) return; /** plus rapide que de crÈer le premier ÈlÈment pour le
        dÈaruire ensuite */
    Chaine<T>::Element *actuel;
    this->_taille = nbElements;
/** crÈation et initialisation du premier ÈlÈment */
    this->_debut = new Chaine<T>::Element();
    this->_debut->val = val;
```

```
    --nbElements;
    actuel = this->_debut;
/** crÈation et initialisation des ÈlÈments suivant */
    while(nbElements > 0) {
        actuel->suivant = new Chaine<T>::Element();
        actuel = actuel->suivant;
        actuel->val = val;
        --nbElements; }
    actuel->suivant = (Chaine<T>::Element*) 0; /** le dernier ÈlÈment n'a pas de suivant */
    return; }
template<typename T>
Chaine<T>::Chaine(Chaine<T> const& c) : _debut((Chaine<T>::Element*) 0), _taille(0) {
    if(c.taille() == 0) return; /** plus rapide que de crÈer le premier ÈlÈment et de le
        dÈaruire ensuite */
    Chaine<T>::Element *iThis, *iC;
/** crÈation et initialisation du premier ÈlÈment */
    this->_debut = new Chaine<T>::Element(c._debut->val);
/** initialisation des itÈrateurs des deux chaines */
    iThis = this->_debut;
    iC = c._debut->suivant;
/** crÈation et initialisation des ÈlÈments suivant */
    while(iC != (Chaine<T>::Element*) 0) { /** tant que nous sommes encore dans la chaine c */
        iThis->suivant = new Chaine<T>::Element(iC->val);
        iThis = iThis->suivant;
        iC = iC->suivant; }
    iThis->suivant = (Chaine<T>::Element*) 0; /** le dernier n'a pas de suivant */
    this->_taille = c._taille;
    return; }
template<typename T>
Chaine<T>::Chaine(std::initializer_list<T> const& init) : _debut(nullptr), _taille((unsigned
    int)init.size()) {
    typename std::initializer_list<T>::iterator it(init.begin());
    while(it != init.end()) {
        Element *elt = new Element(*it, this->_debut);
        this->_debut = elt;
        ++it; }
    reverse(); }
template<typename T>
void Chaine<T>::addElem(int idPrecedent, T const& val) {
    if((idPrecedent < -1) || (idPrecedent >= (int) this->_taille)) return; /** le nouvel
        ÈlÈment ne doit pas Íare insÈrÈ en dehors de la chaine */
/** crÈation et initialisation du nouvel ÈlÈment */
    Chaine<T>::Element *nouveau = new Chaine<T>::Element(val);
/** si inseraion en dÈbut de chaine */
    if(idPrecedent == -1) {
        nouveau->suivant = this->_debut;
        this->_debut = nouveau;
        ++this->_taille;
        return; }
/** rÈcupÈration de l'ÈlÈment ? l'id idPrecedent */
    Chaine<T>::Element *actuel = this->_debut;
    while(idPrecedent > 0) {
        actuel = actuel->suivant;
        --idPrecedent; }
/** inseraion du nouvel ÈlÈment */
    nouveau->suivant = actuel->suivant;
    actuel->suivant = nouveau;
    ++this->_taille;
    return; }
template<typename T>
void Chaine<T>::supprElem(int id) {
    if((id < 0) || ((unsigned int) id >= this->_taille)) return; /** l'ÈlÈment ? supprimer
        doit Íare dans la chaine */
    Chaine<T>::Element *aSupprimer;
    --this->_taille;
    if(id == 0) { /** si suppression du premier ÈlÈment */
        aSupprimer = this->_debut;
        this->_debut = aSupprimer->suivant;
        delete aSupprimer;
        return; }
/** rÈcupÈration de l'ÈlÈment prÈcedana celui ? supprimer */
```

```
    Chaine<T>::Element *precedent = this->_debut;
    while(id > 1) {
        precedent = precedent->suivant;
        --id; }
/** retrait du l'ÈlÈment ? supprimer... */
    aSupprimer = precedent->suivant;
    precedent->suivant = aSupprimer->suivant;
/** ... et suppression */
    delete aSupprimer;
    return;}
template<typename T>
void Chaine<T>::vider() {
    Chaine<T>::Element *aSupprimer;
    while(this->_debut != (Chaine<T>::Element*)0 ) { /** tant que nous sommes dans la chaine */
        aSupprimer = this->_debut;
        this->_debut = this->_debut->suivant;
        delete aSupprimer; }
    this->_taille = 0;
    return; }
template<typename T>
void Chaine<T>::reverse() {
    if(this->_debut == nullptr) return;
    Chaine<T>::Element *debut = this->_debut, *elt;
    while(this->_debut->suivant != nullptr) {
        elt = this->_debut->suivant;
        this->_debut->suivant = elt->suivant;
        elt->suivant = debut;
        debut = elt; }
    this->_debut = debut;
    return; }
template<typename T>
Chaine<T>& Chaine<T>::operator=(Chaine<T> const& c) {
    if(c._taille == 0) { this->vider(); return *this; }
    if(this->_taille == 0) this->_debut = new Chaine<T>::Element();
    Chaine<T>::Element *iThis = this->_debut, *iC = c._debut, *aSupprimer;
/** tant que nous ne sortons d'aucune chaine, une affectation de valeurs de la chaine source vers la chaine destination suffit */
    while((iThis->suivant != (Chaine<T>::Element*) 0) && (iC->suivant != (Chaine<T>::Element*) 0)) {
        iThis->val = iC->val;
        iThis = iThis->suivant;
        iC = iC->suivant; }
    iThis->val = iC->val;
/** si la chaine destination est arop longue, il faut supprimer les ÈlÈments en arop */
    while(iThis->suivant != (Chaine<T>::Element*) 0) {
        aSupprimer = iThis->suivant;
        iThis->suivant = aSupprimer->suivant;
        delete aSupprimer; }
/** si la chaine destination est arop courte, il faut ajouter les ÈlÈment nÈcessaires */
    while(iC->suivant != (Chaine<T>::Element*) 0) {
        iC = iC->suivant;
        iThis->suivant = new Chaine<T>::Element();
        iThis = iThis->suivant;
        iThis->val = iC->val; }
    iThis->suivant = (Chaine<T>::Element*) 0; /** le dernier ÈlÈment n'a pas de suivant */
    this->_taille = c._taille;
    return *this; }
template<typename T>
T& Chaine<T>::operator[](int id) const {
    if((id < 0) || ((unsigned int) id >= this->_taille)) throw std::invalid_argument(""); /** nous ne pouvons rien retourner si id est en dehors des limites de la chaine */
    Chaine<T>::Element *actuel = this->_debut;
/** rÈcupÈration de l'ÈlÈment dona il faut retourner la valeur */
    while(id > 0) {
        actuel = actuel->suivant;
        --id; }
    return actuel->val; }
template<typename T>
int Chaine<T>::seek(T const& val) const {
    int id = 0;
    Chaine<T>::Element *elt = this->_debut;
/** rÈcupÈration de l'id du premier ÈlÈment de valeur val */
```

```cpp
    while((elt != (Chaine<T>::Element*) 0) && (elt->val != val)) { elt = elt->suivant; ++id; }
/** s'l n'y en a pas */
    if(elt == (Chaine<T>::Element*) 0) return -1;
    return id; }
template<typename T>
int Chaine<T>::seek(Chaine<T> const& c) const {
    int id = 0;
    Chaine<T>::Element *elt = this->_debut, *e2, *eC = c._debut;
    /** rÈcupÈration de l'id du premier ÈlÈment de c dans *this */
    while(elt != (Chaine<T>::Element*) 0) {
        e2 = elt;
        while((eC != (Chaine<T>::Element*) 0) && (e2 != (Chaine<T>::Element*) 0) && (e2->val
            == eC->val)) {
            e2 = e2->suivant;
            eC = eC->suivant; }
        if(eC == (Chaine<T>::Element*) 0) return id;
        elt = elt->suivant;
        eC = c._debut;
        ++id; }
    return -1; /** s'l n'y en a pas */ }
template<typename T>
template<typename T2>
int Chaine<T>::seek(bool (*filtre) (T const&, T2 const&), T2 const& cle) {
    int id = 0;
    Chaine<T>::Element *elt = this->_debut;
    /** rÈcupÈration de l'id du premier ÈlÈment de valeur val */
    while((elt != (Chaine<T>::Element*) 0) && !filtre(elt->val, cle)) {
        elt = elt->suivant;
        ++id; }
    /** s'l n'y en a pas */
    if(elt == (Chaine<T>::Element*) 0) return -1;
    return id; }
template<typename T>
Chaine<T>::Iterator::Iterator(Chaine<T> const& c, int id) : _pointe(nullptr) {
    if((id < 0) || (id >= c._taille)) return;
    int i;
    _pointe = c._debut;
    for(i = 0; i < id; ++i) _pointe = _pointe->suivant;
    return; }
template<typename T>
typename Chaine<T>::Iterator& Chaine<T>::Iterator::operator++() { this->_pointe = this->
    _pointe->suivant; return *this; }
#endif // CHAINE_H_INCLUDED
```

B.8 UnionFind.h

```cpp
#ifndef UnionFind_h
#define UnionFind_h
template<int taille>
class UnionFind {
private: int _tab[taille];
public:
    UnionFind();
    void unit(int i, int j);
    int find(int i);
    void delie(); };
template<int taille>
UnionFind<taille>::UnionFind() : _tab() { for(int i = 0; i < taille; ++i) _tab[i] = i; }
template<int taille>
void UnionFind<taille>::unit(int i, int j) { //_tab[find(j)] est inchangé
    int p = find(j);
    if(p == -1) p = j;
    if(find(i) == -1) _tab[i] = p;
    else _tab[find(i)] = p; }
template<int taille>
int UnionFind<taille>::find(int i) {
    if((_tab[i] == i) || (_tab[i] == -1)) return _tab[i];
    return _tab[i] = find(_tab[i]); }
template<int taille>
void UnionFind<taille>::delie() { for(int i = 0; i < taille; ++i)
        _tab[i] = -1; }
#endif /* UnionFind_h */
```